Insieme-RS
A Compiler-supported Parallel Runtime System

dissertation

by

Peter Thoman

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of doctor of science

advisor: Prof. Dr. Thomas Fahringer, Institute of Computer Science

Innsbruck, July 26, 2013

Certificate of Authorship and Originality

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Peter Thoman, Innsbruck, July 26, 2013

Abstract

The efficient use of modern, highly parallel hardware requires an advanced software infrastructure. Two essential components of such an infrastructure are compilers capable of supporting parallelism and their associated runtime systems. Compilers focus on analysis and optimizations which can be performed statically, independently of program execution. Conversely, traditional parallel runtime systems purely consider information which can be derived during runtime as a basis for their scheduling and work distribution decisions.

The parallel runtime system Insieme-RS presented in this thesis goes beyond the capabilities of the state of the art by offering close integration with a high-level optimizing and analyzing compiler. This integration includes the forwarding of meta-information such as static analysis results from the compiler to the runtime system, and compile-time fine-grained function multi-versioning. Using this meta-information, dynamic knowledge only available during program execution, such as the values of program variables, input data sizes and external system load, can be combined with the results of static compiler analysis to yield better scheduling decisions. Additionally, by means of multi-versioning, aggressive static optimizations can be applied selectively at runtime, depending on dynamic conditions.

As it is a target platform of the Insieme compiler, Insieme-RS can enhance a variety of existing parallel programs designed for OpenMP, Cilk or OpenCL. Within this thesis, three novel methods in the fields of multi-process scheduling, automatic work distribution in parallel loops and the granularity control of task-based parallelism are demonstrated on existing programs, and all outperform existing parallel runtime systems by utilizing the unique capabilities offered by Insieme-RS.

Acknowledgements

Throughout the years of research which culminated in this thesis, several people have supported me greatly. Foremost, I'd like to thank my advisor Prof. Thomas Fahringer for his fundamental role in my doctoral work. His advice, motivation and guidance were instrumental in completing this thesis. I would also like to thank the rest of the thesis committee, and in particular the external reviewers for providing an additional perspective on this work.

The group of people graduate students are most likely to spend the majority of their time with are their fellow students, and for me it was no different. As such, I'm deeply indebted to all former and current Ph.D. students in the Distributed and Parallel Systems group, for making our work together not just productive but also, surprisingly often, entertaining. The members of the Insieme team – Herbert Jordan, Simone Pellegrini, Klaus Kofler, Philipp Gschwandtner, Ivan Grasso, Luis Ayuso, and Ferdinando Alessi – deserve particular thanks for many a stimulating discussion, as well as for their tireless work on the Insieme project – without their dedication much of my research would not have been possible.

Making the decision to pursue a Ph.D. is not something done lightly, and I would like to acknowledge some of those who set and encouraged me on this path. Dr. Armin Landmann and his inspirational way of presenting science had a significant impact on me in school, and I have tried to adopt some of his methods in teaching. Prof. Otmar Scherzer and Dr. Harald Grossauer mentored me during my undergraduate studies and also first introduced me to post-graduate research.

Finally, I would like to thank my friends and family for supporting me over my entire studies, and for putting up with the inscrutable mood swings and strain which may sometimes accompany any research or writing activity. In particular, I'd like to thank my mother for all her support, and my grandmother for her incredible joy in my accomplishments, which in turn motivates me to do my best.

Contents

Chapter 1

Introduction

1.1 Motivation

Over the past decade, the overall landscape of computing - from traditional personal computers to portable and embedded devices - has undergone a marked shift towards parallelism [5]. This development, brought on by physical constraints limiting further increases in sequential processing performance [80], has wide-reaching consequences in many areas of computer science. In particular, developing software which efficiently uses these modern parallel systems is a crucial challenge, both in industry and academia.

A Historical Perspective The issues software developers are faced with when creating programs for modern-day, complex parallel systems are perhaps best understood from a historical context. When high-level programming languages were becoming more widely used in the 70s, computers largely performed sequential processing, and memories were almost as fast as the processors performing operations on their data [89]. Therefore, a developer needed only to consider the performance characteristics of the processor and how his high-level language of choice is translated to that processor in order to write an efficient high-performance program.

However, the *memory gap* (the time it takes, in CPU cycles, to access some information in memory) continued to grow, and new hardware features such as caches were introduced to bridge this gap. Adding new hardware components unavoidably increases the number of factors involved in the performance of a program, and makes that performance harder to understand and predict. Introducing *instruction-level parallelism* (ILP) [46], out-of-order execution and branch prediction [71] added more layers of complexity. This architectural evolution made it very hard for programmers to fully utilize or even understand the hardware architecture of a given target platform, and optimizing compilers gained a more important role in program development.

When the rate of sequential performance improvements dwindled further, additional levels of parallelism were introduced in hardware. This included *single-instruction multiple-data* (SIMD) processing in the form of vector units, and *multiple CPU cores* on a more coarse-grained level. For the first time, this type of full-scale parallelism was available not just in the form of distributed nodes in a high-performance computing (HPC) cluster, but on wide-spread consumer hardware. At the same time, programming these HPC clusters became even more complex, as inter- and intra-node parallelism required different approaches [21].

Finally and most recently, *heterogeneity of compute resources* has become an important topic [18]. Different types of algorithms and tasks are suited for different types of hardware approaches, which makes combining multiple types of hardware resources an attractive approach. While there have been some hardware platforms designed specifically to accelerate compute tasks (e.g. the Cell Broadband Engine Architecture [25]), the most widely adopted form of mainstream heterogeneous computing is GPU computing [66], which makes use of graphics processing units (GPUs) to accomplish compute tasks.

Taking into account all these developments, a modern HPC system may consist of multiple nodes connected by a network, each of which features multiple levels of parallelism (sockets, cores, hardware threads, vector units and ILP), a complex memory hierarchy, and perhaps even additional accelerator hardware which has its own set of hardware peculiarities and requirements. Clearly, making efficient use of such an architecture to solve a given problem is harder than ever before.

The Software Challenge Given this level of hardware complexity, software challenges arise in two distinct areas: the required development effort for high-performance programs, and the runtime software support to actually interface with a given hardware platform. Compilers are one of the primary means for tackling the former, while runtime systems enable the latter.

The software development challenge is most easily understood by looking at the technologies and standards a developer needs to be familiar with in order to implement a high-performance program. Back at the start of the historical timeline described above, knowing a programming language such as Fortran or C and the characteristics of a rather basic (by modern standards) CPU would be sufficient. Today, a HPC programmer might well be expected to be familiar with OpenMP and pThreads for intra-node parallelism, MPI for inter-node communication, some set of intrinsics (e.g. Intel AVX [34]) to make full use of vector units, and CUDA or OpenCL in order to deal with heterogeneous accelerators. Since it seems infeasible to train a sufficient number of developers to fully understand this cornucopia of standards, *tool support*, as well as new standardized language efforts and libraries are required to mitigate the effect of this explosion in complexity on software development.

Runtime systems are a crucial part of this tool support, however, they also introduce their own set of challenges. Each type of hardware resource and each level of parallelism is traditionally addressed with a different system – for example, OpenMP on the intra-node shared memory level, MPI on the inter-node level and OpenCL for accelerators. However, these systems are not designed to interact with each other, and need to be orchestrated manually. This is a time-consuming process during development, and can also cause inefficiencies during execution due to the interaction of separate systems. However, it is also an unavoidable effort for HPC programs, as the right balance between the individual components and technologies has a significant impact on performance [67].

The Insieme Project As such, software development tools are tasked both with supporting developers in successfully leveraging increasingly parallel hardware, and with easing the interactions between multiple disparate types of hardware parallelism. Two particularly relevant areas of interest in this regard are parallel compilers and their accompanying runtime systems.

Within the *Insieme project* [28] at the University of Innsbruck, a source-to-source parallel compiler infrastructure and accompanying runtime system are being developed to address some of the software challenges posed by such highly parallel and heterogeneous hardware. The *Insieme compiler* supports multiple input languages and standards such as C, OpenMP, Cilk, OpenCL and MPI, with C++ support currently under development. All of these languages are transformed into a unified, input code independent intermediate representation of parallel programs (INSPIRE [43]), which allows for research into new compiler techniques. The compiler also integrates state-of-the-art transformation tools for INSPIRE to enable quick prototyping and experimentation [45].

The output programs generated by the Insieme compiler use the *Insieme Runtime System* (Insieme-RS) to interact with, monitor and dynamically reconfigure hardware, enable and manage parallel execution and perform online code tuning and steering. The Insieme project attempts to leverage the particular opportunities afforded by strong integration between a high-level analyzing and optimizing compiler and an accompanying runtime system. This symbiotic relationship can aid dynamic decision making - such as scheduling - as well as providing opportunities for code specialization by means of multi-versioning. Insieme-RS is intended as a multi-paradigm runtime system, which can leverage parallelism at both coarse and fine-grained levels, support data parallelism as well as (recursive) task parallelism, and enable parallel execution across a wide range of hardware devices.

1.2 The State of the Art

Related work for this thesis generally conforms to one of two distinct categories: fully developed, feature-complete parallel runtime systems, and individual works aiming at one of the particular areas of contribution treated in chapters 4, 5 and 6. The latter are discussed in more detail within their respective chapters, while this section will give an overview of existing parallel runtime systems, and describe how Insieme-RS differs from previous efforts. We have categorized the related work into four subsets: production-quality runtime systems associated with some compiler, entirely new languages and programming models for parallelism, parallel libraries which are used directly by the programmer and do not change the base language, and research-focused parallel runtime systems.

Compiler-associated Runtime Systems Parallel runtime systems of commercial quality are often offered in conjunction with compilers, for example the Intel compiler [40], the runtime library offered by Microsoft Visual Studio, or the GOMP [60] system used by the GNU Compiler Collection in its OpenMP implementation. These systems are very stable and well-supported, but are rarely targeted directly as platforms for research. As these systems are used daily for production codes, they need to closely mirror language standards, and maintain obsolete features for long periods of time. This results in a long-term accumulation of code which makes it harder to experiment with, prototype and research new methods.

The Cilk system [15] started out as a research project, but, since its acquisition by Intel, falls into a similar category as the commercial-quality systems listed above. Furthermore, all of these systems are generally targeted at supporting one particular type of parallelism generated using just one language or standard – e.g. OpenMP. The Cilk runtime system supports Cilk programs, and GOMP as well as the Microsoft OpenMP library support OpenMP programs. Conversely, Insieme-RS was designed specifically to support multiple parallel paradigms as the target platform for the Insieme Compiler, and to allow for productive research by enabling quick prototyping and experimentation with new algorithms, for example in loop or task scheduling.

New Languages and Programming Models Many runtime systems in current research are tied directly to a new language or programming model, which are often introduced in conjunction with their runtime systems. Important examples of this pattern include the following:

- Charm++ [47] is a new object-oriented parallel language based on C++. Charm++ programs consist of a set of message-driven objects called *chares* which communicate via messages. These objects are mapped to processors by a dynamic runtime system, and messages between objects are handled in an asynchronous fashion. The runtime system also supports reliability features for fault tolerance, such as automatic checkpointing.

- The Chapel [22] parallel programming language developed by Cray focuses on separating algorithmic structure from data layout details. This allows for the optimization of data access locality at runtime by changing the placement of computing work depending on the existing data distribution. In terms of data distribution, it adapts many ideas from High Performance Fortran [55].

- X10 [24] is a partitioned global address space (PGAS) language [90] developed at IBM. It features the concept of *places*, which hold data and activities performing computation on that data. These places are mapped to locations in the hardware model. Similarly to Insieme-RS (see Section 2.3.2), X10 features a parent-child relation between activities, which determines their possible execution order.

- StarSs [52] is a general task/node-level programming model based on adding pragma annotations to a base language. It features various extensions and implementations (such as CellSs [13], SMPSs, GPUSs [7] or GridSs) for a variety of different target architectures. While StarSs aims to unify the programming model for these distinct hardware platforms, each such platform has its own optimized runtime system.

- CUDA [61] extends the C language with primitives to launch data-parallel *kernels* on GPU accelerator hardware. In recent versions it has also been extended for dynamic parallelism [62], breaking free of the limitation of strict separation between a host system generating work and an accelerator performing it. However, CUDA is bound to one particular vendor's hardware.

While these systems offer interesting platforms for the exploration of new software engineering ideas, they do not solve the problem of improving the performance of existing programs on new hardware. Because Insieme-RS can be used by all the output programs of the Insieme compiler, it is widely applicable to existing software. Beyond the practical advantage in real-world use of not having to rewrite programs, this also allows for more direct comparability of research results with existing solutions. This can be an issue with entirely new languages which require a new set of input programs, as it may be hard to discern whether performance improvements stem from the new language features, their respective runtime systems, or perhaps simply the implementation choices in the necessarily distinct input programs.

Parallel Libraries All the solutions outlined above are runtime systems which are designed to implement basic language features of a parallel language or are associated with a particular compiler. There exists an entirely separate way of tackling the problem of programming parallelism, which can still encounter many of the same issues (e.g. scheduling and data or work distribution). Parallel *libraries* are included from an existing source language without changing or extending the fundamental language itself, and offer capabilities for parallel programming. While the runtime systems and new programming models outlined previously may also be implemented at least in part as libraries, the following systems are distinguished by being intended to be used directly by developers as libraries.

The most important options in this category, all of which are currently in widespread use, are:

- The POSIX pThreads library [58] is the oldest shared memory parallel processing library which is still highly relevant today. It offers low-level access to the threading and synchronization primitives available in most modern operating systems. While this potentially allows programmers to fully utilize the performance of the hardware, doing so comes at a high cost in terms of implementation effort. Also, unlike higher-level systems, the pThreads implementation itself cannot perform any optimization in scheduling or data distribution – these are all fixed by the program. The low-level access and widespread availability of the pThreads library make it an ideal candidate as a base for implementing higher-level runtime systems, such as Insieme-RS.

- The Message Passing Interface (MPI) [72] is, in many ways, the distributed memory counterpart to pThreads. It also focuses on providing low-level functionality, and as such takes a similar position on the scale of programmer control versus productivity and runtime optimization options. As for pThreads, this makes MPI a good base on which to build higher-level runtime systems.

- OpenCL [76] is to accelerator computing what pThreads and MPI are to shared and distributed memory parallelism, respectively. However, it is distinct from the other library approaches listed here in that it is both a library, and a language standard. The *host programs* which control the processing of an OpenCL application are written in C or C++ and use library routines to execute data- or task-parallel *kernel code*, which can be run on both accelerators and standard multicore CPUs. This code is written in a C-like language which prohibits the use of some problematic C features such as jumps while adding new syntax, particularly for SIMD parallelism and the use of local memory spaces.

- Intel Threading Building Blocks (TBB) [68] are a high-level C++ template library which implements a task-stealing system for parallelism. In addition to scheduling monolithic tasks, the TBB library allows the generation and execution of *algorithmic skeletons* composed of multiple interdependent tasks, which are executed respecting their graph dependencies. The runtime engine of the TBB library tries to optimize for efficient cache usage as well as load balance.

- Microsoft Parallel Extensions [53] are a library of functionality for parallel programming created by Microsoft for their .net family of languages. The extensions comprise *Parallel LINQ*, a concurrent query execution engine, the *Task Parallel Library* (TPL) which implements the thread pool pattern for tasks and offers convenience methods for loop parallelization, and a set of coordination data structures used to synchronize execution.

- Apple Grand Central Dispatch (GCD) [2] implements the thread pool pattern of task parallelism for C, C++ and Objective C. While it offers extensions to those languages to ease the specification of closures, it can also be used without these extensions, which is why we classified it as a library approach. The GCD runtime system schedules lightweight tasks on work queues, which are implemented using OS-level threads. Similarly to Insieme-RS, it also features an event system to asynchronously trigger the execution of tasks.

Although the library approach to parallelization has many advantages – new technologies can be implemented, distributed, and iterated on more quickly than in a compiler or language standard, the base language tools can largely be reused, and shared library improvements can be made even if only a binary distribution of a program is available – it also precludes the compiler from understanding and optimizing parallelism. As research in Insieme-RS focuses on how close integration between a parallelism-aware compiler and a runtime system can be utilized in optimization, a pure library implementation is not possible in our case.

Research-focused Systems There has been a large number of other parallel runtime systems specifically designed for and dedicated to research proposed over the past decades. Often, these are one-off projects that are quickly created to study some particularly interesting behaviour or algorithm, and then dropped shortly after. A few are designed to be used longer and remain more extensible: recent, modern examples include libKOMP [19], a high-performance OpenMP library and X-Kaapi [37], a multi-paradigm runtime system. Insieme-RS distinguishes itself from these systems by its central design principle, which specifically provides for – and in some cases even depends on – compiler-generated metadata related to and multiversioning of fine-grained program fragments.

1.3 Organization

This thesis is structured into 5 major chapters as follows. Chapter 2 introduces a formal descriptive basis for the programs executed by the Insieme runtime system, and for the hardware which they are executed on. The actual runtime system, including the major entities it manages, the set of possible operations upon them and its integration with the Insieme compiler are detailed in Chapter 3. Finally, Chapters 4, 5 and 6 each introduce a single concrete contribution achieved on the basis of Insieme-RS.

All of these contributions use the unique capabilities of our runtime system to solve a problem in parallel computing, and all of them are as or more effective than existing methods while reducing programming complexity for the developer. Chapter 4 deals with the efficient scheduling of multiple separate parallel processes in a shared memory system, Chapter 5 introduces a combined compiler and runtime approach for fully automatic work distribution in parallel loops and Chapter 6 describes a novel method to regulate the granularity of parallel tasks.

Chapter 2

Model

In order to accurately describe the operation of Insieme-RS an understanding of the program model and the static object descriptions as well as dynamic object instances interacting within the runtime system is required. Furthermore, how this program model within Insieme-RS relates to the original input program of the Insieme Compiler needs to be defined. Additionally, the hardware platforms Insieme-RS programs are executed on must be formally specified. This chapter begins with an introduction of this hardware model, and then proceeds to detail the static program model which describes Insieme-RS applications. Finally, the dynamic instantiation of the objects created during the execution of a program within Insieme-RS, and their potential interactions, are formally specified.

2.1 Target Hardware Model

In this section the model used by Insieme-RS to describe and interact with any hardware platform will be presented. This model holds all the information needed to describe the hardware and its specific entities (such as CPU cores or memory blocks) and their properties (e.g. the size of a cache). This information can then be used to identify entities, query their properties, associate performance measurements to specific hardware, or decide on the distribution of data and work items. The model is used to describe the underlying hardware with as much detail as could potentially be required for Insieme-RS. However, it is not necessary for the runtime system to use all the information available. For many tasks, a simpler view of the hardware might be more beneficial. Hence, for specific use cases, the hardware model might be simplified. Furthermore not all properties described here might be available on all physical instantiations of this hardware model but are rather supplied on a best-effort basis, depending on how much information can be gathered from the user and the system.

2.1.1 Hardware Entities

The target hardware model consists of a directed graph $\mathcal{H} = (E_{\mathcal{H}}, C_{\mathcal{H}})$. The set of vertices $E_{\mathcal{H}}$ models hardware *entities* $e_i^{\mathcal{H}} \in E_{\mathcal{H}}$ with $0 \leq i < |E_{\mathcal{H}}|$, which are connected by directed edges $c^{\mathcal{H}} \in C_{\mathcal{H}} \subseteq E_{\mathcal{H}} \times E_{\mathcal{H}}$, whereby an edge $(e_a^{\mathcal{H}}, e_b^{\mathcal{H}}) \in C_{\mathcal{H}}$ represents a directed connection from $e_a^{\mathcal{H}}$ to $e_b^{\mathcal{H}}$. An entity can be an instance of any of the following types:

Cores. The main computational units which can execute some work item on their own without any other computational entities.

Memory blocks. Entities which represent a single, continuous logical address space and hold a single or multiple memory segments.

Memory segments. Sub-entities which are capable of holding user or system data.

Caches. Memory not explicitly accessed by a user program which is used to mitigate access delays to memory segments.

Accelerators. Additional, independent computational units (e.g. GPUs) which are not capable of computing a work item on their own.

Functional units. Special computational units, dependent on and part of cores but possibly shared among multiple cores. Examples are SSE [39] and AVX [34] units on x86 processors, or NEON [3] engines on ARM chips.

Secondary storage. Entities accessed for file I/O or that hold potentially required data which is not present in any memory block.

Scratchpad memories. A – usually small and low-latency – memory segment which is uncached and explicitly accessible by a user program.

Network interfaces. These are entities that form connections between nodes and network attached secondary storage.

The connecting, directed edges – usually representing either a physical bus or a network – between those entities can be read-only, write-only or read-write connections. Since the hardware model reflects the underlying hardware as closely as possible, only entities that have direct connections should be connected in the model (e.g. a core is connected to the cache and the cache to the memory segment, there is no direct connection between the core and the memory segment if the corresponding hardware features a cache in between those elements). Furthermore, these edges are characterized by at least two properties, *latency* and *bandwidth*, which allow Insieme-RS to assess their performance.

Definition 1 (***Latency and Bandwidth***)

The latency $\mathrm{l} : C_{\mathcal{H}} \mapsto \mathbb{R}$ *of a connection* $c^{\mathcal{H}} = (e_0^{\mathcal{H}}, e_1^{\mathcal{H}})$ *is defined as the time, in seconds, required to transfer a minimal amount (usually one byte) of data from* $e_0^{\mathcal{H}}$ *to* $e_1^{\mathcal{H}}$*. Similarly, the bandwidth* $\mathrm{b} : C_{\mathcal{H}} \mapsto \mathbb{N}$ *specifies the number of bytes that can be transferred from* $e_0^{\mathcal{H}}$ *to* $e_1^{\mathcal{H}}$ *over a period of one second during a sustained transfer.*

Derived Entities

In addition to the concrete entities of the hardware model defined above, we introduce a set of *derived entities*. Each of these represent a set of physical entities, and they are used to allow easier description and depiction of complex hardware models. These derived entities are defined as follows:

CPU. This derived entity comprises a number of cores and (optionally) their connected cache levels, scratchpad memories and functional units.

GPU. A GPU consists of a memory block, an accelerator and its associated caches and scratchpad memories.

Node. Potentially featuring instances of all the concrete entities outlined previously, a node represents one computer system which may be connected via its network interface to other nodes.

SMMP. A shared-memory, multiprocessor system featuring multiple CPUs and memory segments, but only a single memory block.

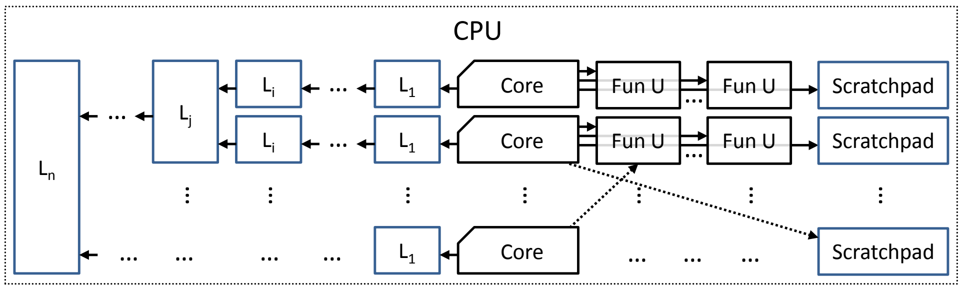


Figure 2.1: Hardware model for the *CPU* derived entity.

Figure 2.1 depicts the potential configurations of concrete hardware entities which may form a *CPU* derived entity. Note that in this and all future illustrations of the hardware model, connections between entities which are already covered transitively are not illustrated by a seperate arrow, to improve readability. The CPU consists of any number of nodes, each potentially featuring a varying number of exclusive or shared levels of cache, functional units and scratchpad memories. The dotted arrows are used to indicate that functional units and scratchpads need not be connected to only a single core, but may also be shared among multiple cores. A recent example of this is the AMD "Bulldozer" architecture [20], in which modules comprising two cores each share a single vector unit.

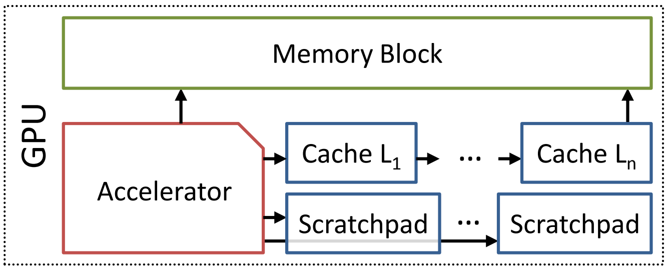


Figure 2.2: Hardware model for the *GPU* derived entity.

The structure of the *GPU* derived entity is illustrated in Figure 2.2. Each GPU consists of an accelerator, its optionally associated caches and scratchpad memories and a memory block representing the GPU global memory pool. Note that unlike CPUs, most modern GPUs offer the possibility of direct, uncached memory access to this pool. This is reflected in the hardware model.

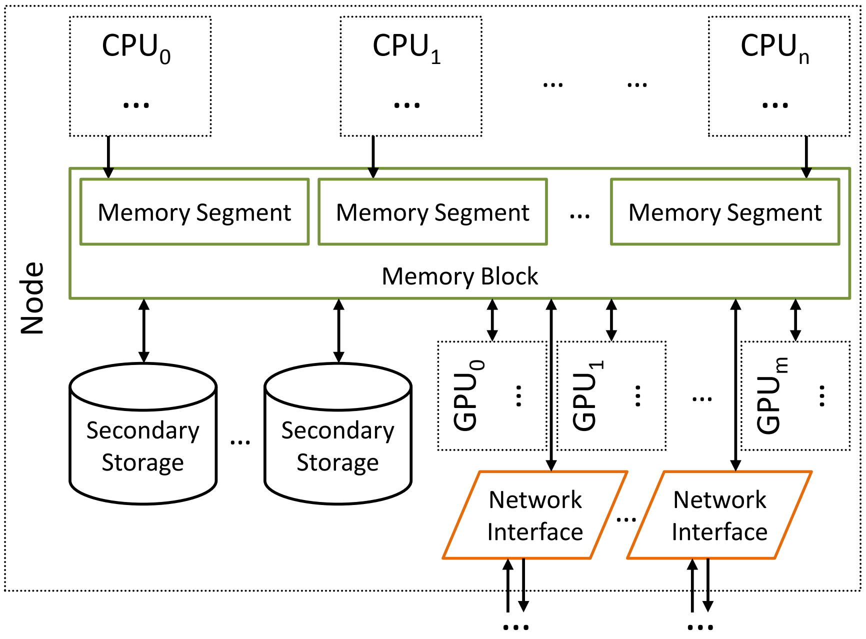


Figure 2.3: Hardware model for the *node* derived entity.

Finally, a *node* derived entity, as depicted in Figure 2.3, encompasses potentially multiple CPUs, GPUs, secondary storage devices and network interfaces, all using the same main memory address space and thus memory block. This memory block may however be split into several segments, e.g. when modeling a NUMA architecture. In this figure, connections to derived entities such as GPUs and CPUs are shown: these are merely a visual aid, representing actual connections to concrete entities within the respective CPUs and GPUs.

Example Instance

Any instantiation of the Insieme-RS target hardware model must consist of at least of one core and one memory block containing a single memory segment. Other entities such as additional memory segments, caches, accelerators, functional units, secondary storage devices and scratchpad memories are optional.

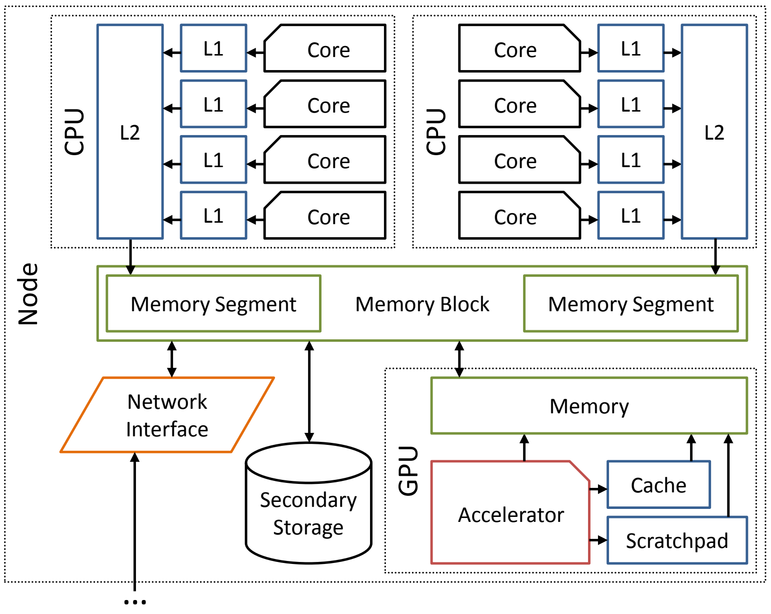


Figure 2.4: Target hardware model example.

Figure 2.4 illustrates a simplified instance of this hardware model for a typical shared memory node comprising two quad-core CPUs and an accelerator. The node is connected to a network interface, which may further connect to more nodes to describe a distributed memory cluster.

All the concrete hardware entities listed above and their general properties and inter-relationships will be described in detail below.

Cores

A core is a computational unit capable of executing some work item on its own with no other computational entity involved. Examples are the cores of any current general purpose CPU. The computational units in most current GPUs and accelerator hardware are excluded, as they require the cooperation of some external CPU core to orchestrate and initiate the computation.

Possible connections of a core and their semantics are listed below. Note that some of these connections may only be realized transitively, e.g. a modern CPU core is likely to be connected to a memory block only via an intermediate hierarchy of caches.

- **Functional units**. Machine code executed on this core will make use of its connected functional units. The most relevant type of functional unit are vector units, which allow for the execution of specifically tuned work items.
- **Caches**. The core uses this cache to read and/or write (according to the properties of the connection) data stored in memory.
- **Memory segments**. The core manages this memory segment and all memory requests to this memory segment need to be performed via this core.
- **Scratchpad memory**. The core has direct access to this scratchpad memory and can read/write (according to the property of the connection) data from/to it.
- **Network interfaces**. The core can access this network interface to send or receive data from other nodes or storage.

Cores are further described by these properties:

- **Architecture**. Denotes the instruction set architecture (ISA) of this core. Examples include x86_64, PowerPC64 and ARMv9.
- **Frequency range**. Describing the clock frequency range that this processor can operate at, which can be used for power/energy saving purposes.
- **Number of hardware threads**. The number of independent threads the core can execute simultaneously in hardware. Usually just one hardware thread is supported, but e.g. IBM's Power 7 cores support up to four hardware threads each.

Memory Blocks

A memory block consists of at least one, but possibly more memory segments which constitute a single, continuous address space. Therefore, a memory block can spread over multiple memory controllers in hardware, with possibly varying access latency properties for individual memory segments. Memory blocks have the follwing properties:

- **Memory size**. Denotes the total amount of memory available in this memory block, in bytes.
- **Set of memory segments**. The memory segments contained in this memory block.

Memory Segments

A memory segment is an entity with a continuous address space, capable of holding data items so that they can be managed by the runtime system. In hardware each memory segment is managed by a single memory controller. Caches are not modeled as memory segments, since the runtime only has very limited direct control over their operation. They hold the following additional properties:

- **Memory size**. Denotes the total amount of memory available in this memory segment, in bytes.

Caches

Caches cannot be directly addressed or managed by the runtime system, however, since their properties have a significant impact on the performance of a processor and the effectiveness of various optimization strategies, these properties are reflected in the hardware model.

Caches hold connections to memory segments, with the semantics that they cache accesses to those memory segments. Cache coherency information is implicitly contained in the way individual cores are (transitively) connected to caches. Multiple caches may be connected to each other to model a cache hierarchy. The following set of properties characterizes caches:

- **Cache size**. Denotes the total amount of memory available in this cache, in bytes.
- **Line size**. Describes the length of a single cache line, in bytes.
- **Associativity**. A number representing how many different locations inside of the cache can be used to store a single memory location, which has an impact on its performance and how replacements are performed. A direct mapped cache would be represented by an associativity of one.

- **Replacement policy**. The replacement policy used to decide which cache lines to drop.

Accelerators

Accelerators are computational units which are managed by Insieme-RS but are not capable of executing any work items on their own. They can only do so with the assistance of cores. Examples for accelerators are graphics processors (e.g. Nvidia Kepler [62]), the SPEs on an IBM Cell processor [25] or correspondingly configured FPGAs.

Accelerators can be connected to scratchpad memories, caches and memory segments with the following semantics:

- **Scratchpad memory**. Accelerators can read/write (according to the properties of the connection) data stored in this scratchpad memory.

- **Caches**. Accelerators can use this cache to read/write (according to the properties of the connection) data stored in memory segments connected to the cache.

- **Memory segments**. Accelerators can directly read/write (according to the properties of the connection) data stored in this memory segment.

The following additional properties describe accelerators:

- **Type**. The type of accelerator, which determines the type of programs that can be executed on it.

- **Frequency range**. Denotes the clock frequency range that this accelerator can operate at, which can be used for power/energy saving purposes.

Functional Units

Functional units are special units associated with a core that are not managed by Insieme-RS but used directly by the code that is executed (e.g. SIMD units used by SSE code on x86 hardware). Such units may be shared among multiple cores. Functional units are only used by cores, and are not connected to any other entity.

Properties:

Vector width. For vector units, denotes the register width that can be operated on, in bits.

Secondary Storage

Secondary storage holds data or work items which are not loaded into memory in such a way that they can be managed by the runtime. Furthermore it might be used for additional data operations like file I/O processing. Since secondary storage is a passive entity it does not hold any connections.

Scratchpad Memory

Scratchpad memory is uncached memory with a contiguous address space that can be accessed randomly by either cores or accelerators (e.g. the local storage in various OpenCL devices [76]). It can be either shared amongst multiple computational units or be exclusive. Scratchpad memory is a passive entity, it does not hold any connections.

Properties:

- **Size**. Denotes the total amount of memory available in the scratchpad memory, in bytes.

Network Interface

Network interfaces are entities used for inter-node communication and for accessing network-attached secondary storage. Network interfaces can be connected to other network interfaces or secondary storage. The – possibly complex – network structure between two network interfaces or network interfaces and secondary storage is not mapped precisely at this point. Instead, a node A holds a single connection to each network interface of all other nodes accessible by node A and all secondary storage entities (if accessible by node A). However, this simplified model can still adequately capture important parameters relevant to runtime decision making processes, such as the relative latency between nodes, as each connection between network interfaces provides its own latency and bandwidth parameters.

Addressing

Every entity $e_i^{\mathcal{H}}$ in the hardware model may be addressed using a triple $(n, T_{\mathcal{H}}, i)$, consisting of the node n this entity belongs to, a type identifier $T_{\mathcal{H}}$ (e.g. accelerator, memory block, ...) and a unique identifier i over all entities of type $T_{\mathcal{H}}$ present on that node in the hardware model.

Edges are addressed as tuples $(e_i^{\mathcal{H}}, e_j^{\mathcal{H}})$ connecting two identified entities, which are directional from e_i to e_j. The addressing scheme itself does not prohibit any connection between arbitrary entities of the hardware model, however, only the connections defined above for each type of entity are semantically correct.

For example, $e_0^{\mathcal{H}} = (0, \texttt{core}, 0)$ would address the first CPU core in node 0 of the hardware model, while $e_1^{\mathcal{H}} = (0, \texttt{cache}, 0)$ would address the first cache in the same node. $c_0^{\mathcal{H}} = (e_0^{\mathcal{H}}, e_1^{\mathcal{H}}) \in C_{\mathcal{H}}$ implies that this CPU core is connected to the cache. A value of $l(c_0^{\mathcal{H}}) = 5 * 10^{-9}$ would mean that the latency for the core accessing this cache is 5 nanoseconds, and $b(c_0^{\mathcal{H}}) = 21474836480$ would imply that 20 Gigabytes of sustained transfer per second are possible over this connection.

2.2 Static Program Model

Insieme-RS is designed to run parallel programs on hardware architectures such as those described by the model introduced in Section 2.1. All such programs are required to conform to a parallel program model which will be detailed in this and the following section.

The program model is split into two separate parts, distinguishing between static, descriptive objects and active, dynamic objects. The former are those which have a directly identifiable counterpart in the program code, while the latter are instantiated during the execution of a program and exist in memory at that point. This section will provide a model for the static objects and structures which exist before the execution of the program, and remain unchanged throughout its execution.

2.2.1 A Simple Sequential Program Model

As a basis for further discussion in this thesis, we will now provide a simple descriptive framework for sequential programs. This model is based on the control flow graph of a program and will subsequently be extended to support parallelism and other advanced features.

Note that we will not define the underlying statements and expressions which could be used in conjunction with this model to fully describe a program, or introduce a complex visibility concept for variables. This representation was deliberately chosen to be as simple as possible, while being sufficient for the purpose of accurately describing the semantics of our runtime system.

Definition 2 (*Sequential Program*)

$$\begin{aligned}\mathcal{P} &= (\mathcal{C}, V)\\ \mathcal{C} &= (S, E)\end{aligned}$$

A program $\mathcal{P}$ comprises a directed graph $\mathcal{C}$ defined on a set of program statements S, and a set of variables V. The directed edges $e \in E$ define the possible control flow between the statements $s \in S$, which form the nodes of the graph. In particular, $e_0 = (s_a, s_b) \in E$ means that there is a possible control flow from s_a to s_b.

In each program $\mathcal{P}$, there is exactly one $s_s \in S$ for which the indegree (i.e. the number of edges $(s_i, s_s)\ \forall s_i \in S$) $deg^{-}(s_s) = 0$. This statement s_s is called the *entry point* of the program $\mathcal{P}$. There can be multiple nodes $s_e \in S$ for which the outdegree (i.e. the number of edges $(s_e, s_j)\ \forall s_j \in S$) $deg^{+}(s_e) = 0$, all of these are called *exit points* of the program.

A central concept in our program model are *regions*. They will be used as building blocks for more complex objects throughout this chapter, and are defined as follows.

Definition 3 (*Program Region*)

A subset $R \subseteq S$ is called a single-entry single-exit program region within $\mathcal{P}$ iff the following conditions hold:

$$\begin{array}{lllll} \exists r_e \in R & \forall r \in R & \forall o \in S \setminus R & : & (o, r) \in E \rightarrow r = r_e \\ \exists r_x \in R & \forall r \in R & \forall o \in S \setminus R & : & (r, o) \in E \rightarrow r = r_x \end{array}$$

The statement r_e is then called the entry node of the single-entry single-exit program region R, and r_x is called the exit node.

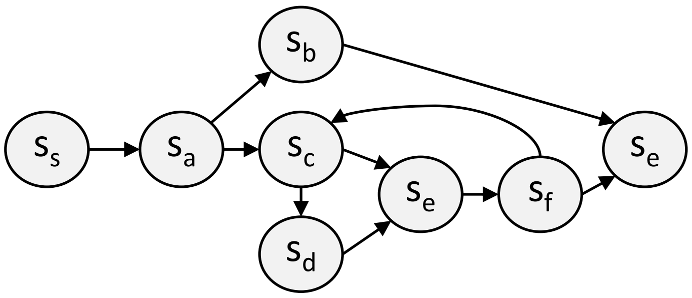


Figure 2.5: Simple sequential program control flow graph.

Figure 2.5 depicts the graph $\mathcal{C}$ for a simple program $\mathcal{P}$. Its entry and exit points are labeled s_s and s_e, respectively. The subset $R = \{s_c, s_d, s_e, s_f\}$ is a single-entry single-exit program region in $\mathcal{P}$, with the entry node s_c and the exit node s_e.

$$v \quad = \quad (\tau, n, s_z) \tag{2.1}$$

Each variable $v \in V$ is defined by a type τ, a dimension n, and an n-dimensional tuple s_z which defines its size. Scalars are defined with $n = 0$, consequently, s_z for such a variable will be the empty tuple. For example, $(\texttt{float}, 2, (10, 10))$ would define a 2-dimensional 10 by 10 array of floating point numbers, while $(\texttt{int}, 0, ())$ defines a scalar integer value.

For each statement $s \in S$, we define two functions which determine the relation between such a statement and the set of variables V.

$$\begin{array}{lll} \mathbf{r} & : & S \times V \mapsto (R_l, R_u) \cup \bot \cup \top \\ \mathbf{w} & : & S \times V \mapsto (W_l, W_u) \cup \bot \cup \top \end{array} \tag{2.2}$$

The function $\mathbf{r}$ captures the read accesses to variables within a given statement, while $\mathbf{w}$ captures write accesses.

R_l and R_u are n-tuples signifying the lower and upper bounds of the orthotope accessed within an array, respectively. If there are no accesses then

$\perp$ is returned, and $\top$ is used to signal an access to a scalar. Definition 2.3 clarifies the semantics of the **r** function, and **w** is defined equivalently for write accesses.

$$\begin{aligned}
\mathbf{r}\left(s, v=(\tau, n, s_z)\right) &= \begin{cases} \perp, & \text{if } v \text{ is not accessed in } s \\ \top, & \text{if } v \text{ is read in } s \text{ and } n=0 \\ \left((r_{l1}), (r_{u1})\right), & \text{if } v \text{ is read in } s \text{ and } n=1 \\ \left((r_{l1}, r_{l2}), (r_{u1}, r_{u2})\right), & \text{if } v \text{ is read in } s \text{ and } n=2 \\ \ldots \end{cases} \\
\mathbf{w}\left(s, v=(\tau, n, s_z)\right) &= \begin{cases} \perp, & \text{if } v \text{ is not accessed in } s \\ \top, & \text{if } v \text{ is written in } s \text{ and } n=0 \\ \left((w_{l1}), (w_{u1})\right), & \text{if } v \text{ is written in } s \text{ and } n=1 \\ \left((w_{l1}, w_{l2}), (w_{u1}, w_{u2})\right), & \text{if } v \text{ is written in } s \text{ and } n=2 \\ \ldots \end{cases}
\end{aligned} \tag{2.3}$$

2.2.2 Parallel Extensions

To extend Definition 2 for parallel programs, the set S of nodes in the control flow graph needs to be extended, and new semantics need to be defined for the additional nodes and edges between them.

Definition 4 (*Parallel Program*)

$$\begin{aligned} \mathcal{P}^p &= (\mathcal{C}^p, V) \\ \mathcal{C}^p &= (S^p, E, P) \\ S^p &= S \cup \Psi \cup X \cup \Gamma \end{aligned}$$

The definition of a parallel program $\mathcal{P}^p$ closely mirrors the sequential definition $\mathcal{P}$, but with three new node types (Ψ, X, Γ), and a new edge type (P), with the following semantics:

- *$\psi \in \Psi$ nodes represent parallel* ***spawn*** *operations. These nodes feature a single outgoing edge $e \in E$, which we call the* sequential control flow, *and another outgoing edge $p \in P$ which we call the* parallel control flow. *Unlike for other nodes, where only a single outgoing edge is taken on a path,* all *edges $(\psi, \_) \in E \cup P$ are taken, at the same time.*

- *$\chi \in X$ nodes represent parallel* ***communication*** *operations. An edge $(\chi_a, \chi_b) \in E$ represents communication between the nodes χ_a and χ_b which are being executed in parallel.*

- *$\gamma \in \Gamma$ nodes represent parallel* ***join*** *operations.* All *edges $(\_, \gamma) \in E \cup P$ need to be taken before proceeding through an outgoing edge of γ.*

For edges in E, exactly one is taken by a path during the sequential execution of a program. Conversely, edges in P may be taken 0 to $n \in \mathbb{N}$ times in parallel. *Edges in P are further restricted to only be of either the type (ψ, r_e) or (r_x, γ) for any entry node r_e or an exit r_x, that is, the parallel operation needs to be a single-entry single-exit code region.*

Each join operation γ_a *corresponds to* a spawn operation ψ_a according to the following principle: let E^{-1} be the inverse relation to E, that is $(s_a, s_b) \in E \rightarrow (s_b, s_a) \in E^{-1}$, and P^{-1} the inverse relation to P in similar fashion. Then ψ_a is the first $\psi \in \Psi$ encountered when proceeding transitively along E^{-1} and P^{-1}. The correspondence between any given ψ_a and γ_a is uniquely defined due to the fact that each parallel operation is a single-entry single-exit code region.

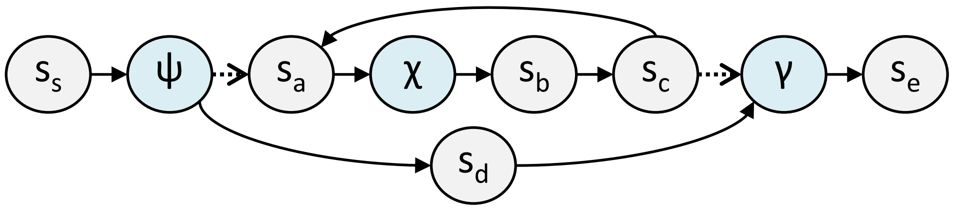


Figure 2.6: Simple parallel program control flow graph.

Figure 2.6 depicts the graph $\mathcal{C}^p$ for a simple parallel program $\mathcal{P}^p$. Its entry and exit points are labeled s_s and s_e, respectively, and dashed arrows represent elements of P while full arrows represent elements of E. The single-entry single-exit region $\{s_a, \chi, s_b, s_c\}$ is executed in parallel with the statement s_d.

At this point, we would will define some common terms related to parallel programs, which will be used throughout the remainder of this thesis.

Speedup Let T_1 be the sequential execution time of a parallel program region $\mathcal{P}^p$, and T_n the execution time of the same region executed with n-fold parallelism. The *speedup* $\mathbf{S}_n$ is then given by $\mathbf{S}_n = \frac{T_1}{T_n}$.

Ideal Speedup When a speedup of $\mathbf{S}_n = n$ is obtained for a code region, it is said to exhibit linear or *ideal speedup*.

Scalability The *scalability* of a code region describes how its execution time varies with the degree of parallel execution n. The scalability of a code region is considered good if the achieved speedup remains close to ideal even for high n.

2.2.3 Modeling Nondeterministic Choice

The sequential and extended parallel program models presented so far relate to existing work in the area of parallel computing. However, they are insufficient to represent a central feature of the Insieme compiler and runtime system, namely the ability to specify nondeterministic choice between different code regions with equivalent semantics. We call these code regions *implementation versions* and model a parallel program with nondeterministic choice $\mathcal{P}^n$ according to Definition 5.

Definition 5 (*Parallel Program with Choice*)

$$\begin{aligned}\mathcal{P}^n &= (\mathcal{C}^n, V)\\ \mathcal{C}^n &= (S^n, E, P)\\ S^n &= S \cup \Psi \cup X \cup \Gamma \cup \Theta\end{aligned}$$

The addition of nondeterministic choice only requires one additional node type compared to the parallel program $\mathcal{P}^p$ in Definition 4. Nodes $\theta \in \Theta$ represent nondeterministic **choice***. All outgoing edge $(\theta, \_) \in E$ are required to model the same semantics – that is, regardless of which outgoing edge is chosen, the input/output behaviour of the program will remain the same.*

The addition of θ nodes allows for different implementation versions of the same functionality to be modeled as separate sub-graphs within $\mathcal{C}^n$.

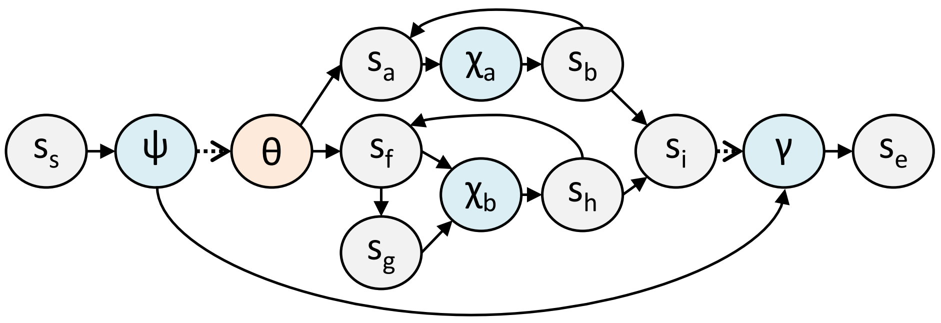


Figure 2.7: Simple parallel program control flow graph with nondeterministic choice.

Figure 2.7 illustrates the graph $\mathcal{C}^n$ for a simple parallel program with nondeterministic choice $\mathcal{P}^n$. As before, dashed arrows represent members of P while full arrows represent members of E. The single-entry single-exit regions characterized by the sets of nodes $\{s_a, \chi_a, s_b\}$ and $\{s_f, \chi_b, s_g, s_h\}$ are semantically equivalent, and the node θ enables nondeterministic choice between them.

2.2.4 Work Item Descriptions

All computations which constitute the execution of a program by Insieme-RS are organized using *work items*. Work items describe a single-entry single-exit, potentially parallel code region within a program, which can be independently scheduled and executed by the runtime system. We distinguish between *work item descriptions*, which are passive, static descriptions of computations as they are represented in the program source code generated by the compiler backend, and *work item instances*, which are objects actively allocated in memory during the execution of a program.

Each work item description $\overline{w} \in \overline{W}$ is an aggregate, structured representation of a single-entry single-exit code region, its parallel properties, resource requirements, potentially multiple implementation versions and additional meta-information. The total number of work item descriptions $|\overline{W}|$ is known statically and remains constant throughout the entire execution of a program. However, every work item description may be instanced multiple times, as described in Section 2.3.2 which deals with work item instances.

Definition 6 provides a formal definition of a work item description $\overline{w}$.

Definition 6 (*Work Item Description*)

$$\begin{array}{rcll} \overline{w} & = & (I, v_w, q) & \\ i & = & (R^n, c, \mathbf{m}) & i \in I \quad c \in \{\text{True}, \text{False}\} \\ \mathbf{m} & : & K^m \mapsto V^m & \end{array}$$

A work item description $\overline{w}$ consists of a set of work item implementation versions *I – with each implementation $i \in I$ realizing the same program semantics, but potentially differing in non-functional behaviour (e.g. execution time, memory usage, parallelism or power consumption), a variable v_w which will be used during program execution to identify individual parallel instantiations of the work item description, and the* resource requirement function *q.*

Each implementation version i further comprises the following components:

- *The executable single-entry single-exit program region R^n it encapsulates, which may use v_w to implement diverging parallel control flow.*
- *A boolean value c indicating whether fully parallel execution is required. If this value is set to* True, *the runtime system is not allowed to partially sequentialize the execution of the individual parallel control flows launched based on this work item description.*
- *The partial function* **m** *representing optional meta-information associated with the implementation. It maps a generic set of keys K^m to values V^m and is described in more detail in Section 3.4.3.*

For example, the work item description

$$\overline{w_0} = (\{i_0 = (R^n_0, \text{False}, \{\}), i_1 = (R^n_1, \text{False}, \{\text{schedule} = \text{dynamic}\})\}, q_0)$$

could represent a parallel loop with two separate implementation versions, which are expressed by the single-entry single-exit regions R^n_0 and R^n_1. Note that c is set to False in both versions, which matches the general semantics of a parallel loop – i.e. individual iterations can be executed either in parallel or may be partially sequentialized. The implementation version i_1 features additional meta-information, in the form of the key/value mapping "schedule = dynamic", which could indicate to the runtime system that the iterations of this loop should be scheduled dynamically.

A value of $c =$ False is generally advantageous in terms of performance, as the runtime system has more potential control over the exact degree of parallelism used. However, in some cases it is not possible to allow such flexibility while maintaining program semantics, for example if the parallel code region R^n encapsulated by the work item contains communication operations (χ nodes).

Ranges of Parallel Operations

While not part of the definition of a work item description (as they are only actively instantiated during program execution), it is still important to note that each work item description can be launched for a range of parallel operations (l, u) numbering between some lower bound l and an upper bound u. This way, the semantics of a ψ node within a parallel program can be mapped to a work item, as a variable number of parallel control flows can be spawned. For each such control flow, the variable v_w will be used to store a separate identifier. These ranges and their semantics are detailed in Section 2.3.2, where work item instances are treated. This section also provides code examples and illustrates the mapping of common parallel structures to work items and operations on them.

Resource Requirement Function

$$q : \mathbb{N}^2 \times I \to 2^{V \times \mathbb{N}^* \times \{RO,RW,WO\}} \times 2^{\mathcal{H}} \tag{2.4}$$

The work item resource requirement function q is detailed in Definition 2.4. It maps a range of parallel operations (l, u) and a particular implementation to its resource requirements, which can include access to data item sub-ranges and particular hardware entities. In this definition, $\mathcal{H}$ is the set of special accelerators or functional unit entities as defined in section 2.1 potentially available within the system, and an instance of the powerset $2^{\mathcal{H}}$ of $\mathcal{H}$ as returned by q represents the specific hardware components required to process the given range and implementation version.

The structure $V \times \mathbb{N}^* \times \{RO, RW, WO\}$ describes some sub-range of a variable in V, and specifies either read-only (RO), read-write (RW) or write-only (WO) access. Its powerset $2^{V \times \mathbb{N}^* \times \{RO,RW,WO\}}$ thus captures any number of accesses of any type to any variables or their sub-ranges.

For example, in

$$\begin{aligned} q\left((0,10),i_0\right) = \Big(\; \{ \; & (v_0,[0],RW), \\ & (v_7,[0\ldots 10, 0\ldots 10],RO), \\ & (v_9,[0\ldots 10],WO)\big\}, \\ \{ \; & e_0^{\mathcal{H}} = (0,ACC,0)\big\}\Big) \end{aligned}$$

the value of $q\left((0,10),i_0\right)$ indicates that executing the range $(0,10)$ of work item implementation version i_0 will require read-write access to the scalar value stored in variable v_0, read-only access to a 10-by-10 sub-region of the two-dimensional array in v_7 and write-only access to a sub-range of the one-dimensional array in v_9. Additionally, it will require access to the hardware model entity $e_0^{\mathcal{H}}$, which is of the accelerator type.

The resource requirement function is generated by the compiler based on the the functions $\mathbf{r}$ and $\mathbf{w}$ defined in Equation 2.2. They are evaluated for every variable v and each statement s in R^n, and the results are gathered to form the powerset $2^{V\times\mathbb{N}^*\times\{RO,RW,WO\}}$.

2.3 Dynamic Program Model

Most of the objects (work items, data items, and communication groups) and interaction between them which make up a parallel program running on Insieme-RS do not exist statically within the program source code. Rather, they are dynamically instantiated during execution, often in ways that are impractical to capture in a static model – such as recursive task parallelism or the dynamic, demand-based distribution of work or data across hardware resources. This section describes the dynamic objects which form our program model, including their formal definition, any potential operations on or between them and their dynamic state.

2.3.1 Program State

Describing the semantics of operations within the dynamic program model requires a concept of the *program state* of a parallel program with nondeterministic choice as specified in Definition 5.

Definition 7 (*Program State*)

$$\begin{aligned} \hat{s} &= \left(H, \hat{V}\right) \\ h &= (s_0, s_1, \ldots, s_N) \qquad \forall i \in [0,N] : s_i \in S^n \\ \hat{V} &= \{v_0 = e_0, v_1 = e_1, \ldots, v_k = e_k\} \end{aligned}$$

The state $\hat{s}$ of a program is formally defined by a tuple with two components:

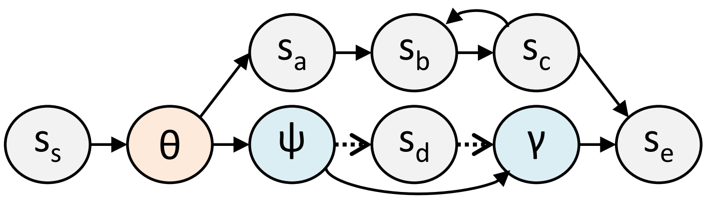
- *A set of paths $h \in H$ on the extended control flow graph $(S^n, E \cup P)$. All the paths in H must be valid in the given program $\mathcal{P}^n$, thus for each h: $\forall i \in [0, N-1] : (s_i, s_{i+1}) \in (E \cup P)$.*
- *$\hat{V}$, a set of pairs of variables v_i and their current values e_i. For each program variable $v_i \in V$ there exists exactly one pair $v_i = e_i$ in $\hat{V}$. For each variable v_w identifying a parallel instantiation (see Definition 4), there exists a tuple of values with each entry corresponding to one parallel path.*

Note that a program state $\hat{s}$ as per Definition 7 captures the entire execution path of each active program-level thread at a given point in time. This is very useful to define the semantics of program constructs, but obviously not feasible in practice, and is not representative of the real implementation within Insieme-RS.

Program State Examples

In order to gain a better understanding of the possible program states allowed by Definition 7 in a parallel program with nondeterministic choice, Figure 2.8 provides an example of such a program. The content of each statement node is provided in a simple pseudo-code notation. Additional information about the spawn operation ψ is also specified: its range of parallel operations is fixed to (v_1, v_1) and the variable used in its work item description to identify the parallel instantiation (see Definition 6) is set to v_ψ. Note that for simplicity, sequences of statements without branching control flow are depicted as a single node (in the case of s_s and s_b) – this is equivalent to a sequence of single-statement nodes with direct sequential connections.

Semantically, the example program performs the initialization of a 4-element array stored in v_2, with one nondeterministic option implementing a sequential initialization (option a), while the other performs the initialization in parallel (option b).



```
ψ: (l,u) = (v_1, v_1)  |  v_ψ
S_s: v_0 = 0; v_1 = 4; v_2 = [0,0,0,0];
S_a: v_3 = v_0;        S_d: v_2[v_ψ] = v_ψ;
S_b: v_2[v_3] = v_3;   S_c: if(v_3 == v_1): s_e;
     v_3 = v_3 + 1;         else: s_b;
```

Figure 2.8: Example program for program state.

We will now define a selection of possible states during the execution of this program, using both the formalism defined above as well as an informal textual description.

- $\hat{s}_s = (\{(s_s)\}, \{v_0 = 0, v_1 = 4, v_2 = [0,0,0,0]\})$. This is the starting state of the program, after the execution of the statements in s_s. A single path exists, which shows that s_s has been executed, and the variables have been given their initial assignment.
- $\hat{s}_{\mathrm{a1}} = (\{(s_s, \theta, s_a)\}, \{v_0 = 0, v_1 = 4, v_2 = [0,0,0,0], v_3 = 0\})$. In this

instance, the edge (θ, s_a) was chosen nondeterministically, and s_a was executed, setting $v_3 = 0$.

- $\hat{s}_{\text{a2}} = (\{(s_s, \theta, s_a, s_b, s_c, s_b)\}, \{v_0 = 0, v_1 = 4, v_2 = [0, 1, 0, 0], v_3 = 2\})$. Continuing from state $\hat{s}_{\text{a1}}$, two iterations of the initialization loop have been executed, setting $v_2 = [0, 1, 0, 0]$.

- $\hat{s}_{\text{a3}} = (\{(s_s, \theta, s_a, s_b, s_c, s_b, s_c, s_b, s_c, s_b, s_c, s_e)\}, \{v_0 = 0, v_1 = 4, v_2 = [0, 1, 2, 3], v_3 = 4\})$. Final state on path a, all four iterations of the initialization loop have been executed, setting $v_2 = [0, 1, 2, 3]$.

- $\hat{s}_{\text{p1}} = (\{(s_s, \theta, \psi)\}, \{v_0 = 0, v_1 = 4, v_2 = [0, 1, 0, 0], v_\psi = (0, 1, 2, 3)\})$. In this instance, the edge (θ, ψ) was chosen nondeterministically, which will initiate parallel execution, providing the tuple of assignments of v_ψ for each parallel path.

- $\hat{s}_{\text{p2}} = (\{(s_s, \theta, \psi, \gamma), h_0 = (s_s, \theta, \psi), h_1 = (s_s, \theta, \psi), h_2 = (s_s, \theta, \psi, s_d, \gamma), h_3 = (s_s, \theta, \psi, s_d)\}, \{v_0 = 0, v_1 = 4, v_2 = [0, 0, 2, 3], v_\psi = (0, 1, 2, 3)\})$. Subsequently to $\hat{s}_{\text{p1}}$, the original program path continued on to the join node γ, and four new parallel paths h_0 to h_3 were generated. The progress of these individual paths is arbitrary – in the example, h_0 and h_1 have not progressed yet, while h_2 has completed (including reaching the join node γ), and h_3 has completed s_d but has yet to reach the γ. This results in an assignment $v_2 = [0, 0, 2, 3]$, with the values at index 2 and 3 written but not those at index 0 and 1.

Note that regardless of which path is chosen in θ, the final value assigned to v_2 when the statement s_e is reached is always $v_2 = [0, 1, 2, 3]$. Thus, the program fulfills the requirement of the θ node that all possible outgoing edges lead to a semantically equivalent execution.

2.3.2 Work Item Instances

While Section 2.2.4 defined work item descriptions, these descriptions are only the static templates from which *work item instances* are generated during the execution of a program. We will now clarify how these instances relate to descriptions, how they are generated and what operations can be performed on them. For improved readability, in the remainder of this document, we use the name "work item" interchangeably with "work item instance", but always refer to work item descriptions by their full name.

Definition 8 (*Work Item Instances*)

$$\begin{array}{rcl} w & = & (\overline{w}, (l, u), t) \\ t & \in & \{\texttt{initializing}, \texttt{ready}, \texttt{running}, \texttt{suspended}, \texttt{resumable}, \texttt{done}\} \end{array}$$

The structure of an individual work item instance is defined above. It consists of the following components:

- *A static, constant work item description $\overline{w}$, as per Definition 6.*
- *An integer range $(l, u) \subset \mathbb{N} \neq \emptyset$, defining the range of parallel operations encapsulated by the work item, and their indices.*
- *A state t, which can take on one of six values, the semantics of which are detailed later in this section.*

The set of all active work item instances W during the execution of a program dynamically forms a directed acyclic graph (W, L) where each node $w \in W$ represents a work item and every edge $(w_1, w_2) \in L$ describes a "launched by" relation between two work items. The full work item graph is not known a-priori. It is established on-the-fly during the execution of each program. Furthermore, note that two different executions of the same static program may result in different sequences of work item execution, and thus graphs (W, L), due to decisions made by the runtime environment regarding the scheduling of work items and the distribution of data items.

Work Item Generation

Within a parallel program with nondeterministic choice $\mathcal{P}^n$ modeled according to Section 2.2.3, a work item can potentially be generated for every single-entry single-exit code region. In case a work item w is generated from a parallel region – that is, one starting with an entry node ψ and ending with an exit node γ – that work item can potentially have a range (l, u) of more than one parallel operation.

In case the work item features a node θ allowing nondeterministic choice, this will be reflected by its work item description $\overline{w}$ providing more than one implementation variant $i \in I$. In particular, $|I| = deg^+(\theta)$ – one implementation variant is generated for each outgoing control flow path from θ.

A node ψ in the original program maps to a work item `spawn` operation, which can launch a work item with either a fixed number of parallel operations or a variable one, depending on the program semantic. A fixed number is represented by supplying an interval (l, u) with $l = u$. To allow individual parallel invocations of the sub-graph R^n representing the code region modeled by a work item w to each perform different operations (it is not usually the goal of a parallel invocation to perform the exact same operations multiple times), the variable v_w designated for the work item in its description will be assigned a different value in each parallel work item instance generated by the spawn operation. More exactly, given $\hat{V}$ as the set of variable assignments prior to passing a ψ node and performing a `spawn` operation, a range of k potentially parallel operations, and the assignment $\hat{V}'$ in each parallel control flow after passing the ψ node: $\hat{V}' = \hat{V} \cup \{v_w = (0, \dots, k-1)\}$.

Semantic Description

Every work item consists of a set of parallel operations (the set may include only a single operation if there is no parallelism) indexed by the interval (l, u) as described in Definition 8 above. Every work item, by way of its associated work item description $\overline{w}$, may have multiple implementations, each of those being a member of $I \neq \emptyset$. The various implementations may target different hardware requirements (e.g. distributed vs. shared memory, CPU vs. GPU, ...), optimization goals or data distribution patterns. The boolean value c determines for the various implementations whether the entire range of parallel operations has to be carried out simultaneously using parallel hardware (e.g. for a message passing based implementation or a shared memory based implementation involving mutual communication via channels or locks). This also requires that the entire index range is split up into smaller fractions in a single step - just before all of the resulting fragments are computed. On the other hand, if $c =$ False for some $i \in I$, the order of computation may be arbitrary. Sub-regions of the index range may be further subdivided, thereby generating a larger number of work items which might be used for improved load balancing.

Every implementation can be invoked for an arbitrary sub-range of the index interval. The resource requirement function function q maps any of the implementations and an arbitrary sub-range of the index set to its corresponding resource requirements, as detailed in Section 2.2.4. The runtime system selects one of the available implementations, satisfies its requirements – allocating the necessary data item instances and provisioning the requested extra hardware, e.g. accelerators – and then starts its execution.

Insieme-RS may chose to process the entire range of the parallel operations covered by a single work-item using small steps. This way, work items can be repeatedly split for load balancing purposes and to generate a varying amount of parallelism. However, it is equally valid semantically to chose to compute the entire work item sequentially. In either case, the runtime system guarantees that every step of the index range is executed exactly once.

During its execution, a work item instance may create new work items. Those *child* work items are handled just like any other work item. In case the parent node has to be able to synchronize (join) with the spawned work items, Insieme-RS needs to maintain the parent/child relationship between these work items.

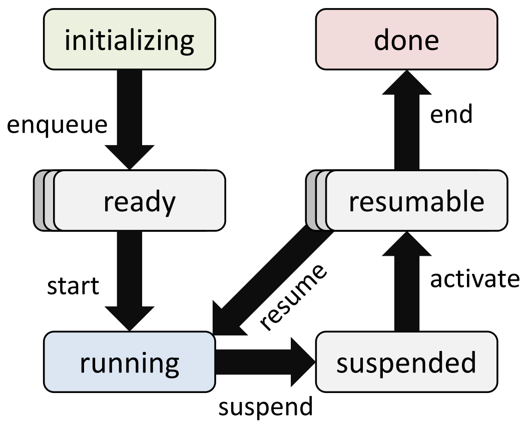


Figure 2.9: Work item states and valid transitions.

Work Item States

Work items can exist in exactly one of the following states:

`initializing` This state indicates that the work item is currently being initialized and is not yet ready to be executed.

`ready` The work item has been placed within a work item queue and is ready to be executed, split or moved to another queue.

`running` The work item is currently being executed by some worker.

`suspended` Execution of this work item has been suspended because of some synchronisation operation, and it will only be resumed after an event has occured which indicates that the condition it was waiting for is fulfilled.

`resumable` This state indicates that the work item is in the work item pool of some worker, and ready to resume execution.

`done` The work item has finished executing.

Figure 2.9 illustrates the possible transitions between these states. Work items in the `ready` and `resumable` states are managed by workers and are thus accessible to scheduling policies, which is indicated in the figure by showing multiple items in a queue.

Work Item Operations

There are a number of operations that may either be performed on work items, or return work items. They often correspond to the state transitions in Figure 2.9.

$w = \texttt{spawn}(\overline{w}, (l, u))$ Create a new work item instance from a work item description $\overline{w}$ and a range (l, u).

$\texttt{enqueue}(w, k)$ Enqueue a work item instance w on some worker k's work item queue.

$W_n = \texttt{split}(w_s)$ Takes a work item $w_s = (\overline{w_s}, (l_s, u_s), t_s)$ and generates a new set of work items W_n, partitioning the range (l_s, u_s) into disjoint subsets while keeping the description $\overline{w_s}$ and state t_s identical. A special variant is the binary_split which generates the following set:

$$W_n = \left\{ \begin{array}{l} \left(\overline{w_s}, (l_s, l_s + \frac{(u_s - l_s)}{2}), t_s\right) \quad , \\ \left(\overline{w_s}, (l_s + \frac{(u_s - l_s)}{2}, u_s), t_s\right) \end{array} \right\}$$

$\texttt{start}(w)$ Start executing a work item. At this point a specific implementation variant $i \in I$ needs to be selected, and the resource requirements need to be evaluated and fulfilled.

$\texttt{suspend}(w)$ Stops a running work item which has not yet finished execution, usually to wait for some event (see Section 3.3.2) or the completion of a group operation (see Section 2.3.4).

$\texttt{activate}(w)$ Generally triggered by some event, this operation is performed on suspended work items to mark them as ready for resuming execution.

$\texttt{resume}(w)$ Resumes the execution of a previously suspended and reactivated work item.

$\texttt{end}(w)$ Ends the execution of a work item, signaling the related event (see Section 3.3.2).

$\texttt{join}(w_i, w_j)$ Suspends the execution of work item w_i until another work item w_j enters its done state.

$\texttt{join\_all}(w_i)$ Suspends the execution of the invoking work item w_i until all work items spawned by this work item enter their done state. This is equivalent to $\forall w_t \in W, (w_i, w_t) \in L^+ : \texttt{join}(w_t)$ with L^+ designating the transitive closure on the "launched by" relation L defined in Section 2.3.2 above.

Work Item Examples

In order to allow for a better understanding of the work item concepts defined above and in Section 2.2.4, we will now provide some examples of typical parallel programming structures, and show how they are modeled using work items. Code will be provided in a C-like pseudocode syntax, utilizing the additional operations and types defined in this thesis.

```
1  $\overline{w_0} = (\{(R_0^n, \text{True}, \{\})\}, q_0)$;
-----------------------------------------------------------
2  w = spawn($\overline{w_0}$, (1,32));
3  enqueue(w, cur_worker());
4  join(cur_wi(), w);
```

Listing 2.1: Fork-join Parallelism

```
1  $\overline{w_4} = (\{(R_4^n, \text{False}, \{\})\}, q_4)$;
-----------------------------------------------------------
2  wl = spawn($\overline{w_4}$, (100,100));
3  enqueue(wl, cur_worker());
4  join(cur_wi(), wl);
```

Listing 2.2: Loop Parallelism

```
1  $\overline{w_1} = (\{(R_1^n, \text{False}, \{\})\}, q_1)$;
2  $\overline{w_2} = (\{(R_2^n, \text{False}, \{\})\}, q_2)$;
3  $\overline{w_3} = (\{(R_3^n, \text{False}, \{\})\}, q_3)$;
-----------------------------------------------------------
4  w1 = spawn($\overline{w_1}$, (1,1));
5  enqueue(w1, cur_worker());
6  w2 = spawn($\overline{w_2}$, (1,1));
7  enqueue(w2, cur_worker());
8  w3 = spawn($\overline{w_3}$, (1,1));
9  enqueue(w3, cur_worker());
10 join_all(cur_wi());
```

Listing 2.3: Task Parallelism

Listings 2.1, 2.2 and 2.3 show basic examples of fork-join, loop and task parallelism. Each listing consists of two distinct parts, separated by a horizontal line: the upper part provides the work item descriptions used in the code in the lower part. These definitions are kept minimal on purpose, with each description providing only a single implementation version, with no meta-information and an unspecified requirement function. The operations `cur_worker()` and `cur_wi()` are used as shorthands to refer to the current worker and work item, respectively.

Fork-Join Parallelism In Listing 2.1 an example of fork-join parallelism is provided. This is functionally equivalent to e.g. an OpenMP `#pragma omp parallel` section. On line 2, a new work item instance `w` is created from work item description $\overline{w_0}$. Note that $c = \text{True}$ in $\overline{w_0}$. This is essential, as the individual parallel execution paths in fork-join programs often communicate with one another, which prevents partial sequentialization. The parallel range for `w` is set to `(1,32)`, which means that Insieme-RS is free to choose any number of parallel instances from a minimum of 1 to a maximum of 32 to execute the region R_0^n. The `spawn` call on line 2 is just an initialization operation: after it, the work item `w` has been created, but no parallel execution has been started yet.

On line 3, this work item is enqueued on the current worker, which is the actual *fork* operation in the fork-join model. From this point onward, multiple parallel execution paths may exist. On line 4 execution on the main path is suspended until `w` has been completed, with the `join` operation directly corresponding in name to the *join* in the fork-join model.

Loop Parallelism Listing 2.2 may seem quite similar to 2.1 at first glance, however, the semantics differ significantly due to the specification of the work item description $\overline{w_4}$ and the parameters of the `spawn` call. The work item `wl` represents a parallel loop with an iteration count of `100`, and thus features a fixed range of `(100,100)`. However, in this case partial sequentialization is allowed (c = False in $\overline{w_4}$), as the semantics of a parallel loop restrict communication within the loop body R_4^n. This type of work item is applicable to modeling all kinds of data-parallel operations, e.g. OpenMP for loops or OpenCL kernel invocations on a work item range.

Task Parallelism Finally, Listing 2.3 shows an example of task parallelism. Three independent work items `w1`, `w2` and `w3` are spawned and enqueued, each with their own work item specification. As they represent tasks, their parallel range is fixed to `(1,1)`, i.e. each one of them is executed by exactly one control flow path. In this case, the value of c in $\overline{w_1}$, $\overline{w_2}$ and $\overline{w_3}$ is in fact irrelevant, as the execution of each work item is already inherently sequential. Note that the `join_all` operation is used to wait for the completion of all the spawned tasks. A variety of task-based parallel programming paradigms can be modeled using this method, including OpenMP 3 tasks and the primitives of the Cilk language.

Summary All three examples are based on the same types (work item descriptions and instances), and use similar operations, but they model entirely distinct parallel behaviour. This illustrates the flexibility of the Insieme-RT program model, allowing it to handle many different use cases with a single representation of parallel work and a comparatively small set of operations.

2.3.3 Data Items

Computation invariably requires data to be applied on, and controlling the location of data elements is one of the key aspects in improving performance on modern architectures. Insieme-RS therefore not only focuses on the scheduling and dynamic optimization of the involved computational tasks, it also takes control of the placement and distribution of data throughout the system. The foundation for memory management within Insieme-RS is the concept of data items – which will be covered in detail within this chapter.

Definition 9 formally defines a data item.

Definition 9 (*Data Items*)

$$d = (\tau, n, s_z)$$

Each data item d in the set of all data items D is considered to be an n-dimensional array of instances of some variable type τ. The shape of the array is required to be an orthotope - hence, the number of elements along every dimension is fixed (the index space is the cross-product of n integer ranges, all starting from 0).

The size of the orthotope is defined by the tuple $s_z \in \mathbb{N}_0^n$. The i-th component of the s_z tuple thereby defines the number of elements along the i-th dimension of the orthotope. If the number of dimensions $n = 0$, the data element represents a single scalar. Consequently, the s_z of such a data item will be the empty tuple.

Note that this definition mirrors the definition of program variables in Section 2. This is by design, as variables are mapped to data items as required to implement the distribution of data in a parallel program.

Semantic Description

Data items are the means by which Insieme-RS manages data access for work items. Whenever a work item w transitions from the `ready` to the `running` state – a transition which only happens once and locks the work item to a specific worker – the resource requirement function q corresponding to the selected implementation i of w is evaluated with the given range of parallel operations (l, u). It returns a set of data items D_r, and for each $d \in D_r$ the required sub-range and type of access (*RO* - read only, *RW* - read write, or *WO* - write only).

Subsequently, the runtime system will process the list of required data item (sub)ranges. Each range which is not yet accessible from the node (as defined by the hardware model presented in Section 2.1) selected for the execution of i will be made available on that node. This is accomplished by allocating memory space in a connected memory segment and copying the data there. Work item w will only be allowed to transition from the `ready` to the `running` state after all its resource requirements are fulfilled.

Sub-Range Example

Let $d = (double, 2, (400, 200))$ be a data item representing a 400×200 matrix of *double* values. Upon creation the full matrix will be allocated on the memory closest to the creating work item. Suppose the matrix should be initialized element-wise with the result of an independent, pure function call. This process can obviously be carried out in parallel. In this case the compiler may decide to perform the operation in parallel, therefore creating a new splittable work item (see Section 2.3.2). If the runtime system decides to exploit this potential parallelism, the initial work item will be sub-divided into smaller work items, each covering only a subsection of the overall matrix (e.g. each covering several rows). The resulting work items will feature a resource requirement function q_M which demands write-only access to a sub-range of the original data item.

$$q_M\left((l, u), i\right) = (d, [l \dots u], WO) \tag{2.5}$$

If the work items are distributed to cores of CPUs mounted on different sockets of a NUMA system, Insieme-RS might allocate a new data item instance, referencing the same variable but representing only a sub-range of the original data item, within the target memory region. Note that in this case no actual data needs to be copied, as only write access is required. As soon as the data environment is initialized, the individual work items can start processing their control flow using the newly allocated memory.

If in a subsequent step the data filled into the matrix needs to be further processed, the already present data distribution may be exploited or sub-ranges may be merged, moved or split to improve the data distribution. Insieme-RS keeps track of the distribution state of every data item. However, for efficiency reasons portions of data items represented by instances need to remain orthotopes.

Data Item Operations

These operations are used within the runtime system to manage and utilize data items:

$d = \texttt{create}(\tau, n, size)$ Creates a data item d of the specified type, dimensionality and size.

$d' = \texttt{access}(d, type, range)$ Allows access to the subrange *range* of d, by means of a new data item d'. The dimensionality of *range* needs to match n of d, the *type* of access is either RO, RW or WO.

$\texttt{free}(d)$ Indicates that d is no longer needed and that the runtime system can reclaim any resources occupied by it.

Data Item Code Example

Listing 2.4 provides a small example of data item usage. A work item representing loop parallelism is used to initialize the entries of an array stored in a data item, with the value at position `i` being set to `i*i`, in parallel. The work item operations utilized in the code are discussed in more detail in Section 2.3.2 and its associated examples.

```
1  $\overline{w_0}$ = ({($R_0^n$, False, {})}, $q_0$);
2  // $R_0^n$ :
3  l = cur_wi()->l;
4  u = cur_wi()->u;
5  dsub = access(cur_wi()->params.dmain, WO, (l,u));
6  for(i=l; i<u; ++i) {
7    dsub->data[i] = i*i;
8  }
9  end(cur_wi());
10 // Main :
11 dmain = create(double, 1, (1000));
12 w = spawn($\overline{w_0}$, (1000,1000));
13 enqueue(w, cur_worker());
14 join_all(cur_wi());
```

Listing 2.4: Data Item Code Example

The listing is split into three parts. In the topmost part, the formal work item definition is specified. As the work item represents a parallel loop, its c value is set to False and it can be partially sequentialized by the runtime system. The next section, starting at line 3, contains the code region R_0^n associated with the work item. In the final section, starting from line 11, the main code region spawning the work item is described.

In line 11 a data item representing a one-dimensional array of 1000 `double` values is created. Write-only access to a sub-range of this array is requested in the work item (line 5). As each parallel instantiation of the work item initializes the values in the range (l, u), exactly that range is requested and iterated over in the loop. Note that the resource requirement function q_0 would be built based on this request, and would take on a similar form as in the sub-range example above (q_M in Formula 2.5).

2.3.4 Communication Groups

In many cases, work item instances are required to exchange information in the form of messages. Furthermore, as it is intended to support multiple parallel programming paradigms, all the communication primitives (e.g. channel and collective operations) generally used in parallel programs need to be supported by the program model of the runtime system.

While channel communication is always a point-to-point communication, where the source and destination is determined by the program itself (by issuing the corresponding commands), collective operations involve an entire set of work items. The runtime system needs to identify the involved work items to realize the requested services.

Definition 10 (*Communication Groups*)

Let W be the set of all work items within the system. At any given time, the set of communication groups $G = 2^W$ is given by a set of subsets of W such that for each $w \in W$ there is a $g \in G$ such that $w \in g$. Furthermore, whenever there is a work item w_i and two communication groups g_1 and g_2 such that $w_i \in g_1$ and $w_i \in g_2$ it follows that either $g_1 \subseteq g_2$ or $g_2 \subseteq g_1$. A work item $w \in W$ is said to be a member *of a group $g \in G$ iff $w \in g$. Finally, g_1 is called a* parent group *of g_2 iff $g_1 \subseteq g_2$.*

Semantic Description

Within a parallel program $\mathcal{P}^n$ modeled according to Section 2.2.3, collective communication operations in χ nodes are always issued to all parallel control flows which have reached a program state by being spawned from the same ψ node. Therefore, Insieme-RS needs to maintain a list of all alive work item instances which belong to such a group sharing the same path, if those groups ever use collective communication. Furthermore, since individual control flows may be members of several such groups, membership in multiple groups has to be supported for work items. However, to reduce the management overhead, we can leverage fact that groups can only be organized in a hierarchical fashion within our program model. This property ensures that if there are two communication groups and one work item instance is a member of both groups, one of the groups must be a subset of the other group.

Internally, communication groups are represented as independent sets, with no explicit relationship between these sets. Work items may join or leave any number of groups by registering respectively unregistering themselves to and from any such set. Whenever a new group is created by some ψ node within a program, a new set of work items representing the groups is also created. It will exist until all of its members have been fully processed. Note that the creation of a communication group can be elided when implementing a ψ node which creates exactly one work item, as is typically the case in task parallelism.

Work items within a communication group g are internally numbered from 0 to $|g| - 1$.

Communication Group Operations

Communication groups are included in the runtime system primarily to allow collective operations to be performed on a group of work items. The following operations are supported:

$g' = \texttt{insert}(g, w)$ Inserts the work item w into the group g, $g' = g \cup \{w\}$.

$g' =$ `remove`(g, w) Removes the work item w from the group g, $g' = g \setminus \{w\}$.

`pfor`$(g, \overline{w_f})$ Distributes the work specified by work item description $\overline{w_f}$ over the work items in group g. This operation needs to be encountered by every $w \in g$.

`barrier`(g) Suspends the invoking work item until every $w \in g$ has invoked this `barrier`.

`join`(g) Suspends the invoking work item until every $w \in g$ has entered its `done` state.

`index`(g, w) Retrieves the index of work item w in group g.

Communication Group Usage

The formalized definition of work groups presented above is simple, but powerful enough to allow the modeling of concepts required for a variety of input languages. OpenMP thread teams launched by a `#pragma openmp parallel` declaration map to one communication group each, with every OpenMP thread inserted into the communication group before being launched, and removed after its completion. Nested parallelism can also be modeled by the concept of parent groups. Operations on OpenMP thread teams, like barriers or pfor operations, can then be mapped to their corresponding communication group. See Section 3.4 for more details.

A similar mapping applies for OpenCL. OpenCL work groups map to communication groups, and the NDRanges used in kernel invocation are parent groups of these communication groups. Synchronisation operations at different levels (ie. the work group level or the whole kernel) can then be mapped to the communication group representing the correct point in the hierarchy.

Chapter 3

The Insieme Runtime System

The aim of Insieme-RS is to provide an environment for the execution of a parallel program specified via INSPIRE and compiled using the Insieme compiler infrastructure. This program may be an entire end-user application or multiple smaller sections of a client application – that is, any application using functionality provided by the runtime system. The runtime system offers a *computation service* to the client application, capable of performing (primarily numerical) computations within a dynamic, heterogeneous, distributed, auto-tuned shared environment. The program sections to be executed within Insieme-RS have to be converted into INSPIRE and customized by the Insieme compiler.

Within the final phase of the compilation process, code interacting with the runtime according to the model covered in Chapter 2 is generated. To handle program execution in a dynamic fashion within Insieme-RS, such a model describing the general structure of processable control flows is essential. Applications are required to obey the rules and restrictions of this model, and the runtime system does not guarantee compensating nor even detecting any violation of the restrictions imposed by the program model.

3.1 Terminology

Within this thesis, some terms unique to the Insieme project are used, and a few common terms in parallel computing deliberately take on a very specific meaning. In this section, all of these terms are defined.

Insieme Components

Insieme Compiler The compiler component developed as part of the Insieme project capable of translating MPI, OpenMP, OpenCL, Cilk and potentially other C/C++ variants to the Insieme Parallel Intermediate Representation (INSPIRE). Furthermore the compiler is capable of analysing and transforming input programs (while applying static optimizations for specific or multiple non-functional parameters). Finally, the compiler's *backend* is responsible for generating target code corresponding to the application model presented by the Insieme-RS API (Chapter 2).

(Compiler) Frontend: part of the compiler capable of translating any supported (parallel) input programs to INSPIRE.

(Compiler) Backend: part of the compiler capable of translating INSPIRE to a specific type of target code. Typically, the target code will again be expressed using some high-level language (e.g. C) and based on additional functionality offered by a runtime environment (e.g. Insieme-RS).

Insieme-RS The Insieme Runtime System (Insieme-RS) is the component, active during execution of a program compiled with Insieme, which manages the hardware infrastructure and provides a model for executing parallel programs. It also covers the dynamic runtime optimization of code and the distribution of computational work among multiple hardware resources. Within this thesis, the expression "*the runtime system*" will always refer to Insieme-RS unless otherwise noted.

INSPIRE The Insieme Parallel Intermediate Representation [43] used to represent programs throughout the system. Although mainly used by the compiler to represent, analyse and transform programs in a uniform way, INSPIRE constructs are also utilized to convey meta-information from the compiler into the runtime.

Software Architecture Related Terms

Insieme-RS Process A single instance of Insieme-RS managing a single shared memory node.

Insieme-RS Environment The network of Insieme-RS processes cooperating across shared memory nodes.

Insieme-RS Client Application A program which was compiled with the Insieme compiler and generated using its Insieme-RS backend. It is built on the parallel program model offered by the runtime system.

Thread A OS-level thread thread of execution maintained by Insieme-RS. Typically, each thread is associated with and bound to a single core of some processor. However, in various cases (Insieme-RS management, IO operations, host programs for accelerators) multiple threads may share the same core.

3.2 Overview

Before providing an in-depth definition of the individual components of Insieme-RS it may prove beneficial to gain an understanding of how these components interact at a high level. This section aims to provide such an overview of the system. Details on all of the components mentioned in this overview are provided within Section 3.3.

3.2.1 Program-level View

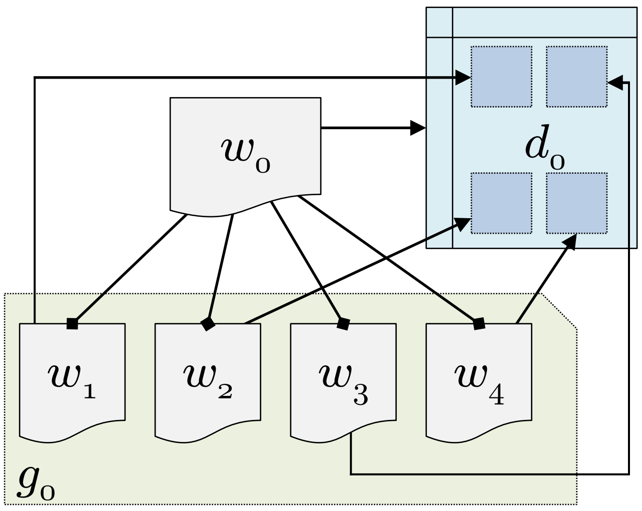


Figure 3.1: Insieme-RS program-level view example.

As a simple starting point, Figure 3.1 illustrates the active runtime objects of a basic program running on Insieme-RS and their relationships. It consists of the following:

- A starting work item instance w_0
- A data item allocated by w_0: d_0
- Four child work items spawned by w_0: w_1, w_2, w_3 and w_4
- Four sub-regions of d_0, each requested by one child work item (using the `access` operation as defined in Section 2.3.3)
- The communication group g_0, containing all four child work items

Note that this application-centric view abstracts from runtime system components such as workers or events – it exclusively contains the objects of the dynamic program model defined in Section 2.3. While workers, events and scheduling policies are essential to the operation of Insieme-RS, they are not required to describe the semantics of a program. Thus, the description of a program in terms of its program model is not only independent of the specific Insieme-RS configuration in use – i.e. in terms of scheduling or event handling – but also does not depend on the underlying hardware. However, both the configuration of Insieme-RS as well as the target hardware platform may have an influence on how the components of the program model are distributed within a system.

Another important observation is that a view of the dynamic objects representing a program, as depicted in Figure 3.1, is merely a snapshot at a particular program state. For example, any work item might allocate a new data item at any point, join or leave a communication group, spawn new child work items or end.

3.2.2 System-level View

From the perspective of the runtime system, the components of a program model – including work and data items as well as communication groups – reside somewhere within the hardware resources managed by Insieme-RS. Work items can be actively executing on a worker (a system-level component managing a thread, defined in Section 3.3.1), ready for execution in any worker's queue or pool, or suspended pending the completion of an event. Data items and their sub-regions are stored in memory blocks, and individual sub-regions might have been moved to optimize access. Both events and communication groups need to be stored separately, and the latter require a list of references to their members.

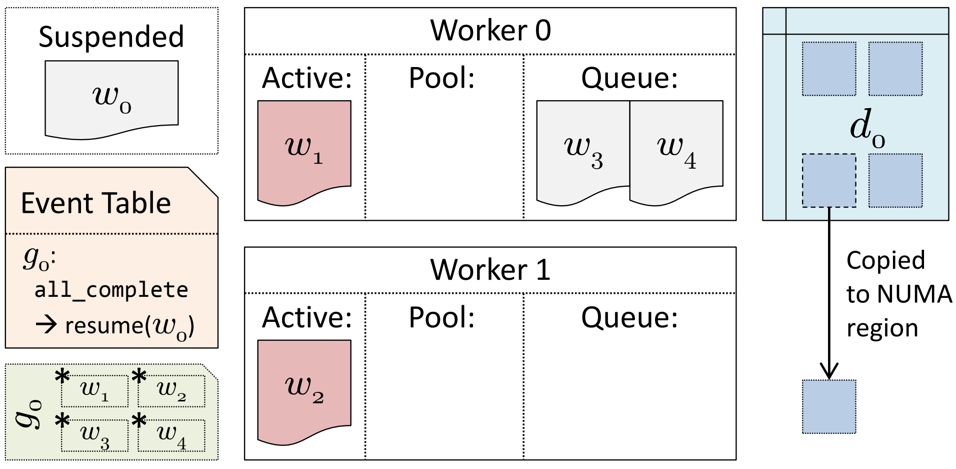


Figure 3.2: Insieme-RS system-level overview.

Figure 3.2 shows a memory-centric, system level overview of the execution of the program-level view illustrated in Figure 3.1. The starting work item w_0 is suspended and waiting for the completion of all items in communication group g_0. There are two workers, each of which is currently executing one item in the group. The remaining two items are depicted as being stored in the queue of worker 0, however, note that this is an arbitrary choice: the semantics would be equivalent were they to be distributed across both workers' queues or both stored in worker 2. The actual manifestation depends entirely on the scheduling policies in use at that point. See Section 3.3.3 for more information.

3.3 System-level Components

The execution of a program within Insieme-RS is defined by the interactions of six major types of objects and processes. The first three of those, which we collectively refer to as the *dynamic program model*, are work items, data items and communication groups. They form the basis for the interaction between any client application and the runtime system by providing a descriptive framework for computation within Insieme-RS, and were introduced in Section 2.3.

All three of these program components – the list of work items, the set of data items and any communication groups – are not known a-priori before executing a program provided by a client application. Essentially, the code generated by the Insieme-RS compiler backend represents a program which will dynamically create work items, allocate data items and form communication groups during its execution. The resulting control flow reassembles the semantics of the input program handed over to the compiler, such that the computational results and IO effects are equivalent to the output generated by the execution of the original program, while allowing Insieme-RS to tune its non-functional parameters.

The second set of objects, including workers, events and scheduling algorithms, describes components and processes which are transparent to the client application, but essential in understanding and describing the behaviour of the runtime system. Together with the program model, this *system model* forms the complete *Insieme-RS model.*

A formal description of the concept of *workers* starts the discussion of these system-level components of Insieme-RS within this chapter. Workers are responsible for all computation in the runtime system and manage the execution and distribution of work items. Section 3.3.1 describes their composition and semantics.

To enable the asynchronous, high-performance tracking of various synchronisation primitives and dependency relations within the program model, *events* are used within Insieme-RS. They allow the execution of arbitrary functions whenever some condition is fulfilled on any object within the Insieme-RS model. As such, they are a powerful tool in implementing the runtime system. They are formally specified and their usage is introduced in Section 3.3.2.

Finally, an important aspect that has a significant impact on the performance of any runtime system are *scheduling policies*. In Insieme-RS, there are actually two separate scheduling problems: the scheduling of work items across workers and the scheduling of collective operations across communication groups. Since both are interesting research topics, their behaviour is based on modular components, which are detailed in Section 3.3.3.

3.3.1 Workers

While work item instances describe the computation carried out by a program, and data items are used to capture their data dependencies, a component is required which actually uses these descriptions to execute a program. In Insieme-RS, workers are entities which actively perform the computation described in work items.

Definition 11 (*Workers*)

$$\begin{aligned} k &= (\mathrm{id}, a, q, p) \\ q &= \{w_0^q, \ldots, w_N^q\} \\ p &= \{w_0^p, \ldots, w_N^p\} \end{aligned}$$

K defines the set of all workers, and each worker $k \in K$ is assigned an id $\in \mathbb{N}_0$ *distributed contiguously starting from* 0*. Thus, if there are $|K|$ workers, they will use* ids *ranging from* 0 *to* $|K| - 1$*. The boolean value a determines whether a worker is active ($a = T$), and is set to $a = F$ for inactive workers. At every point during the execution of Insieme-RS, $\exists i \in [0, N-1] : a_i = T$ is required to hold. The other two components q and p, called, respectively, the work item queue and the work item pool, each represent a set of work items that k is currently managing ownership of. Both $q = \emptyset$ and $p = \emptyset$ are valid.*

Semantic Description

Workers generally correspond to OS-level threads, with at most one worker spawned for each hardware thread available on the system. The responsibilities of workers include generating, distributing and executing work items. To this end, each worker is equipped with two semantically distinct sets of work item instances:

- The **work item queue** q exclusively contains work items in the `ready` state. These work items can be moved freely within the entire Insieme-RS environment, their execution ranges can be split to increase the available parallelism as needed, and the concrete implementation version to be used can still be selected.

- The **work item pool** p, which exclusively contains work items in the `resumable` state. These work items can only be migrated within one Insieme-RS process, and their execution range and implementation have already been fixed to a specific instantiation and can no longer be changed.

This distinction is essential, since, as described in Section 2.3.3, the data environment for each work item is initialized at the point when it transitions from the `ready` to the `running` state. At the same time, a specific implementation variant of the work item is selected. Once this has taken place, it is no longer possible to migrate the work item to a different memory block, as it may have allocated private memory which is not managed by the runtime system.

3.3.2 Events

A common feature in many parallel runtime systems are dependencies which determine the available work at any point during execution. Work items could be waiting on the completion of previous work items on which their results depend, or work items within a communication group might need to be suspended until some collective operation – for example a barrier – has completed. To optimize throughput, these dependencies need to be resolved in an asynchronous, unified manner. The Insieme-RS event system is designed to fulfill these requirements.

Definition 12 (*Events*)

$$\begin{aligned} \epsilon &= (y, t, H) \\ y &\in W \cup D \cup G \cup K \\ h &= (h_f : U \to \{T, F\}, u) \end{aligned}$$

Each event ϵ within Insieme-RS is defined by three components. The source object y, which can be any work item, data item, communication group, or worker, the event type t which semantically depends upon the type of the source object, and the set of event handlers H. In the initial state of every event $H = \emptyset$. Every individual event handler $h \in H$ is composed by a function h_f defining the desired response to the event and some data of arbitrary type u.

Semantic Description

The event system allows any object within Insieme-RS to wait for any other source object to trigger some specific event type, and it supports multiple handlers being registered for every such combination of source object and event types. The set of event types is pre-defined for each type of object, and may e.g. include `started` or `finished` for work items. When an event of type t is triggered for some source object y, the function $h_f(u)$ is invoked for each $h \in H$. Event handlers control their own persistence by means of the return value of h_f, which determines for each h whether it will be included in the new set of event handlers H_{new}:

$$h \in H_{\text{new}} \equiv h \in H \wedge h_f(u) = T$$

Thus, event handler response functions which return T will cause their corresponding event handlers to persist over multiple occurrences of an event, while a return value of F will remove the corresponding event handler after the event has occurred. Note that persistent event handlers are not always semantically meaningful – for example, the "finished" event for a work item will only occur exactly once, so even a persistent handler will never be invoked again.

Event Usage

Events are used internally to model a number of operations of the runtime system. For example, a `join`(w_j) operation issued by a work item w_i can be modeled in a straightforward fashion by first registering the event

$$(w_j, \texttt{finished}, (\texttt{resume}, w_i))$$

and subsequently suspending the execution of w_i. Once w_j transitions to its `done` state, it will trigger its `finished` event, which will in turn execute the `resume` event handler, resuming w_i. Thus the desired semantics of the `join`(w_j) operation are fulfilled.

Event Types

Work items	**Data items**	**Workers**	**Groups**
started	allocated	start_sleep	joined
suspended	moved	woken	left
resumed	deleted		all_complete
finished			
children_finished			

Table 3.1: Event types triggered by runtime objects

Table 3.1 lists the defined event types for each object type within Insieme-RS. The semantics are mostly self-explanatory, with one exception: the work item event type `children_finished` is triggered for some work item w_a when all child work items of w_a, including those launched transitively, are completed – that is, once the condition $\neg\exists w_b : (w_a, w_b) \in L^+$ given the "launched by" relation L as defined in Section 2.3.2 is fulfilled. The event system is modular and additional event types can be defined easily if the need arises.

3.3.3 Scheduling Policies

Unlike the previous sections about work and data items, workers, communication groups and events, which presented objects within the runtime system, this section will describe a *process*, namely scheduling. This necessitates a somewhat different format, however, the scheduling of parallel work is a fundamental task of Insieme-RS, enabling much of the research conducted based on it, and thus merits an in-depth discussion.

There are two separate scheduling processes within the runtime system: work item scheduling and the scheduling of work distribution operations acting on a communication group. Each will be discussed in a separate subsection.

Work Item Scheduling

Work item scheduling includes two largely orthogonal concepts: work item splitting and work item distribution. The former describes how work items with a range of parallel operations (l, u) incorporating more than a single element are handled. The latter determines how work items are distributed across workers.

Work Item Splitting Using the `split` operation on a work item (see Section 2.3.2), multiple work items representing sub-ranges of the original item can be generated. This property can be leveraged in scheduling to improve load balancing by increasing the amount of available parallelism. To this end, three important splitting policies have been defined:

- **No splitting.** All work items are considered to be monolithic, and no splitting is performed.

- **Lazy binary splitting.** Before starting a work item w from a queue q, if its work range is sufficiently large, it is split in two halves w_1 and w_2 by means of the `binary_split` operation as described in Section 2.3.2. w_1 is started while w_2 is enqueued in q. This method allows for good load balancing while maintaining relatively low overhead, due to its hierarchical nature [85].

- **Static splitting.** Every newly created work item w is immediately split into fragments $w_0, w_1, \ldots, w_{N-1}$ with $N = |K|$ representing the number of workers in the runtime system. This method is particularly well suited to regular programs which feature an inherently balanced load distribution.

Work Item Distribution For the distribution of work items among workers there are two methods, which can either be combined or used separately. The first is the *push* method, which involves a source worker actively selecting a target and moving a work item to it. Conversely, in the *pull* method, every worker only moves work items from another worker to itself. The most common distribution policies employing these methods are the following:

- **Static distribution.** Generated work items are pushed to workers in a round-robin fashion. This works well for balanced workloads in combination with a static splitting policy, as it introduces very low overheads. However, in the case of imbalanced workloads this policy does not allow the runtime system to perform any dynamic load balancing.

- **Random stealing.** Generated work items are initially stored in each worker's local queue. When a worker with an empty queue tries to schedule a work item, it selects a random target worker and tries to pull one work item from its queue. The exact behaviour of this policy depends on whether this pull operation takes an item from the head or tail of the target queue.

Both the splitting behaviour and the work item distribution policy are exchangeable components in Insieme-RS, and multiple versions of each are implemented and directly comparable. In Chapter 6, the random stealing distribution policy is extended to support a novel task optimization approach.

Work Item Scheduling Examples We will now provide two examples of common combinations of work item splitting and distribution policies, and how they affect the execution of work items. In both examples, there are 4 workers $k_0 \ldots k_3$ and the situation at the start is that worker k_0 has one large, splittable work item while all others are empty. To simplify the graphical representation, the queue and pool for each worker are not separated.

Figure 3.3 illustrates the scheduling process in case a static splitting policy is combined with static, push-based distribution. The initial work item w is immediately split into four fragments by k_0. Subsequently, k_0 pushes each of these fragments to a worker in round-robin fashion. Only after all fragments are distributed does k_0 start executing its own fragment

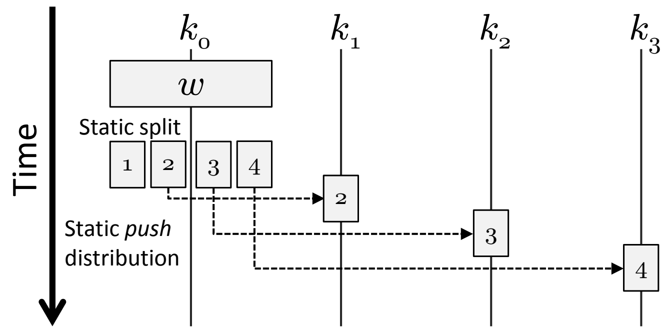


Figure 3.3: Static work item splitting and distribution.

0. The other workers can start executing their assigned fragments as soon as they receive them.

While Figure 3.3 shows a very static scenario which always proceeds in exactly the same manner, Figure 3.4 depicts a far more dynamic set of policies, and thus represents only one possible out come of many. Here, lazy binary splitting is combined with random stealing as the work item distribution method. When trying to start w, k_0 determines that it is sufficiently large, and splits it into two halves. The second half is immediately stolen by k_2 (this is an arbitrary choice). Both halves are still too large to execute without further splitting, so both k_0 and k_2 split their fragment, at roughly the same time. Subsequently, fragment $1a$ is executed by k_0, and $2a$ by k_2. The other workers k_1 and k_3 steal $1b$ and $2b$, respectively (this choice is again arbitrary).

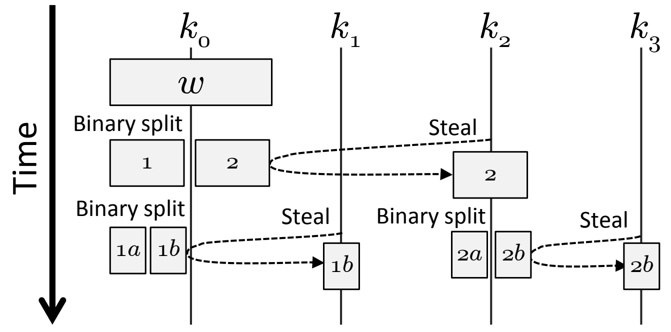


Figure 3.4: Lazy binary work item splitting and stealing distribution.

Work Distribution (pfor) Scheduling

The `pfor(`g`,` $\overline{w_f}$`)` operation was defined on communication groups in Section 2.3.4 and distributes the work range described by $\overline{w_f}$ across the set of work items $\{w_0, \ldots, w_{|g|-1}\}$, which are members in group g. Thus, the work items already exist and have started executing in their own data environments, and are in the `running` state at the point where they encounter the `pfor` operation. This differs significantly from the work item scheduling as described above, which operates exclusively on work items in the `ready` state.

This scheduling operation is therefore equivalent to partitioning an interval $[l, u) \subset \mathbb{N}$ into $|g|$ disjoint subsets, each of which will be assigned to one work item in g. This is similar in concept to loop scheduling in OpenMP [65], and can use many of the existing OpenMP loop scheduling methods. The scheduling strategies available in Insieme-RS include:

- *Static* scheduling, with an optional chunk size parameter c. If the chunk size is not set, all member of g will be assigned equally sized chunks (as far as possible, if $c = (l - u)/|g|$ is not an integer). For a given c, chunks are distributed as follows: $[l, l + c) \rightarrow w_0, [l + c, l + c * 2) \rightarrow w_1, \ldots, [l + c * (|g| - 2), u) \rightarrow w_{|g|-1}$

- *Dynamic* scheduling, also with an optional chunk size parameter c. In this case, the interval is again split into chunks of equal size c, which are distributed to participating work items on a first-come-first-served basis. The default value for c is 1

- *Guided* scheduling. Here, the chunk size parameter takes on a different role, as the size of the distributed chunks is not kept constant throughout the execution of a given loop. Chunks are distributed dynamically on a first-come-first-served basis as in dynamic scheduling, but their size is progressively decreased. The chunk size parameter specifies a minimum size.

- *Fixed* scheduling is used by dynamic optimizers in Insieme-RS to directly manage which sub-ranges are assigned to which work item. For a group of $|g|$ work items, $|g| - 1$ cutoff points $\mathrm{cu}_0, \mathrm{cu}_1, \ldots, \mathrm{cu}_{|g|-2}$ need to be provided. The loop is then distributed as follows: $[l, \mathrm{cu}_0) \rightarrow w_0, [\mathrm{cu}_0, \mathrm{cu}_1) \rightarrow w_1, \ldots, [\mathrm{cu}_{|g|-2}, u) \rightarrow w_{|g|-1}$

- *Balanced* scheduling is also intended for use by optimizers. It differs from fixed scheduling in how the ranges for each work item are specified. While fixed scheduling uses cutoff points, in the balanced scheduling strategy a value $0 \leq s_i \leq 1, s_i \in \mathbb{R}$ is provided for each work item w_i, which specifies the relative *share* of work to be assigned to that item. Compared to fixed scheduling, this has the advantage of being applicable to varying loop sizes, but causes slightly higher overhead and is not as convenient to use for some dynamically adjusting optimizers. This scheduling policy was used in work relating to the automated distribution of OpenCL workloads across devices [50].

The *dynamic* and *guided* scheduling strategies require communication between the work items participating in the execution of the `pfor`, while this is not the case for the other methods. A hybrid, automatic loop scheduling algorithm which selects from these methods based on work item meta-information and information about the current system state is presented in Chapter 5.

3.4 Compiler Integration

An essential feature of Insieme-RS is its close integration with the Insieme Compiler. This enables many of the research opportunities investigated in later chapters of this thesis, and sets it apart from other runtime systems. In this section, the methods by which this integration with the compiler is achieved are presented. To allow for a more complete understanding of the entire process, we start out by briefly outlining the way OpenMP programs are treated in the Insieme compiler and translated to INSPIRE. Subsequently, the mapping from INSPIRE constructs to the Insieme-RS program model, which takes place in the compiler backend, is presented. Finally, we detail how the primary means of integration between the compiler and runtime system – work item multiversioning and the transport of meta-information – are accomplished. A complete introduction of the INSPIRE language is beyond the scope of this thesis and can be found in a separate publication [43].

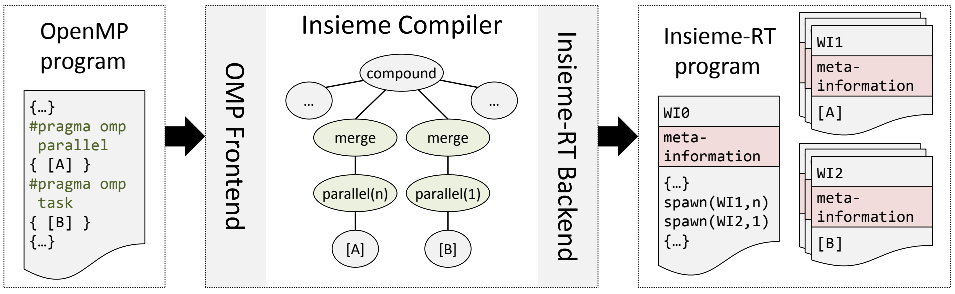


Figure 3.5: Overview of compiler pipeline for Insieme-RS.

Figure 3.5 depicts a simplified version of the pass of an OpenMP program through the compiler. The OpenMP frontend translates the input program into an INSPIRE DAG, which is in turn transformed into a set of work item descriptions by the Insieme-RS backend. Work items are annotated with meta-information, and work item descriptions representing parallel computation (`WI1` and `WI2` in the example) may feature multiple implementation variants.

3.4.1 OpenMP in the Insieme Compiler

The vast majority of the experimental evaluation in this thesis is based on OpenMP programs. OpenMP is a widely used industry standard for parallel programming, and most OpenMP constructs map easily to the internal representation used within the Insieme compiler, INSPIRE. This section describes how the OpenMP frontend of the Insieme compiler maps common structures to INSPIRE, which is a required first step towards running them on Insieme-RS. Note that Insieme at the current point in time supports only a subset of the OpenMP 3.2 standard, however this subset is sufficient to correctly execute a large number of real-world benchmarks and codes.

The `parallel` Construct

The `parallel` construct is the most essential part of OpenMP, as it generates the parallelism used by all other aspects of the language. In Insieme, the code fragment:

```
#pragma omp parallel
baseLangStatement;
```

is translated to the following INSPIRE code:

```
group g = spawn(1, INT_MAX) {
default {
    initLocalDataEnv
    baseLangStatement'
    teardownLocalDataEnv
}
merge(g)
```

The details of implementing the various optional clauses which may be attached to the parallel constructs, as well as the specifics of how local data environments are constructed are omitted for clarity, for this and all the following descriptions. The Insieme compiler supports the full range of shared, private and reduction operations by implementing their semantics in INSPIRE (within the `initLocalDataEnv` and `teardownLocalDataEnv` blocks indicated in the listing above), capturing references to variables in the surrounding context, creating local substitutes within the job body (generating the modified `baseLangStatement'`) and initializing them accordingly. However, the resulting code is relatively complex and unnecessary for the discussion in this thesis.

Note that the INSPIRE thread group `g` is immediately merged, implementing the OpenMP fork-join model of parallelism whereby the thread reaching the parallel block is suspended until the whole block has been processed.

Worksharing Constructs

All the OpenMP work sharing constructs – `for`, `single`, and `sections` – are encoded based on the INSPIRE $pfor$ operator using the iterator range of the associated loop (in the case of `for`), a single iteration (for `single`), or one iteration per `section`. An additional *barrier* is appended in the absence of a `nowait` clause on the worksharing construct.

Tasks and Taskwait

OpenMP tasks are modeled using *spawn* with a fixed range of 1, resulting in the creation of an INSPIRE thread group consisting of a single thread. Since the Insieme parallel model allows for recursively nested parallelism, tasks may spawn additional tasks as required. The `taskwait` primitive available for joining tasks in OpenMP is directly translated into a call to the $mergeAll\,()$ function in INSPIRE.

Synchronisation

OpenMP barriers, whether explicitly provided in the source code or implicit in the semantics of a worksharing construct, are equivalent to the INSPIRE call

$$redistribute\,(unit, (array\,\langle unit\rangle\;\; data)\,\{return\ unit;\}) \tag{3.1}$$

where *unit* is the constant representing the one instance of the *unit* type.

Other means of synchronization in OpenMP, including critical regions and locks, are modeled using INSPIRE channels of the type $channel\,\langle unit, 1\rangle$, which implement locking semantics based on their single-element buffer. Since channel operations in INSPIRE lock when their buffer is full, and a channel of type $channel\,\langle unit, 1\rangle$ has only a single-element buffer, it can be used to model a mutually excluding lock as follows. Acquiring the lock is equivalent to sending an empty message into the channel (thus occupying its buffer), which will block further messages (i.e. lock operations) from being executed. Releasing the lock is accomplished by reading the empty message from the channel, emptying its buffer and allowing another locking operation to take place.

Atomic operations are realized using the generic INSPIRE *atomic* function.

Other OpenMP Features

Widely used OpenMP features which require special consideration when implemented in Insieme are `threadprivate` variables. These are represented in the compiler by allocating a *vector* containing a copy for each thread at the entry point of the program, and forwarding a reference to this vector to each function which uses the threadprivate variable. The vector is indexed using the INSPIRE $getThreadId(0)$ function, which, in combination with the `parallel` implementation outlined above, matches the required OpenMP semantics.

The `omp_get_thread_num` family of OpenMP library functions is implemented by mapping the calls to corresponding calls to $getThreadId$.

3.4.2 INSPIRE → Insieme-RS Mapping

`INSPIRE`	`Insieme-RS`
thread	work item instance
N-d *vector*	N-d data item
threadgroup	communication group
$spawn(1, 1)$	use the work item `spawn` operation to instantiate a single work item
$spawn(a, b)$	create a communication group and multiple work items in that group
$pick(...)$	create and run a work item, with the work item description featuring multiple implementation versions corresponding to the options in pick
merge	`join` (either on work item or group, depending on context)
mergeAll	`joinAll`
$pfor$	`pfor` (need to generate a work item description for the body)
redistribute	`barrier` if it is of the form in declaration 3.1, `redistribute` otherwise
$getThreadID(N)$	implemented using the `index` operation on the corresponding communication group the current work item is a member of
atomic	mapped to corresponding atomic methods if the hardware offers them, otherwise uses locking

Table 3.2: Mapping from INSPIRE to Insieme-RS.

In the Insieme-RS backend of the compiler, INSPIRE structures need to be mapped to objects of the Insieme-RS program model. As the runtime system was designed to complement the compiler, this is usually a straightforward process, but it is fundamental in understanding how INSPIRE programs are executed on Insieme-RS. Table 3.2 summarizes the mapping from INSPIRE types and operators to Insieme-RS objects and operations on these objects.

Note that in the cases where there is no direct one-to-one mapping, the generalized representation in INSPIRE is mapped to several distinct object or operations in Insieme-RS. This is expected and consistent with the goals of each of these components of Insieme: INSPIRE aims to provide a minimal unified representation to simplify analysis and transformation, while Insieme-RS requires specialized code which optimizes for the unique use case at hand.

3.4.3 Work Item Generation

A unique feature of Insieme is the close integration between compiler and runtime system. The primary module where this integration manifests itself is within the Insieme-RS backend of the compiler, where work item descriptions are generated from INSPIRE code. As defined in Section 2.3.2, work item descriptions can feature any number of semantically equivalent implementations of a work item, and each of them can have arbitrary meta-data associated with it. This section describes how these values are derived from the INSPIRE representation and analysis performed by the compiler. Note that while this explanation remains on a more abstract level, Section 3.5.2 details the actual implementation of the mechanisms described here.

Work item descriptions are generated by the Insieme-RS backend whenever it translates either a *spawn* or a $pfor$ operation from INSPIRE. First, the sub-graph representing the code to be executed within the generated work item is outlined, and the required parameters are gathered in a new data item structure. Subsequently, the resource requirements of the code within the sub-graph are determined, and the resource requirement function is generated accordingly.

Work Item Multi-Versioning

Tunable non-functional parameters are supported in INSPIRE by means of the *pick* operator, defined as $(\mathit{list}\,\langle\alpha\rangle) \rightarrow \alpha$. Its semantics are simple: $\mathit{list}\,\langle\alpha\rangle$ is required to be a list of options of an arbitrary, but consistent, type, which are all semantically equivalent in terms of the functionality of the program. That is, replacing a call to $\mathit{pick}\,(\mathit{list}\,\langle\alpha\rangle) \rightarrow \alpha$ with any item in $\mathit{list}\,\langle\alpha\rangle$ is required to yield a correct program, which in turn means that the selection of a specific element of the list can be tuned to optimize for some non-functional parameter.

Work item multi-versioning is thus accomplished in INSPIRE by using the *pick* operator within a *spawn* or $pfor$ operation to select from a $\mathit{list}\,\langle\alpha\rangle$. Here α represents some function type matching the signature required by the surrounding context. The Insieme-RS compiler backend can then generate a set of work item implementations I with $|I| = \mathit{length}\,(\mathit{list}\,\langle\alpha\rangle)$.

In this set of implementations, each element $i = (R^n, c, \mathbf{m}) \in I$ corresponds to one option within the list $list\,\langle\alpha\rangle$. The control flow for to the single-entry single-exit region R^n and the concurrency flag c are generated individually for each separate INSPIRE sub-graph corresponding to each option in the list.

Meta-Information

While the generation of single work items and work items with multiple implementation versions has now been covered, the partial function $\mathbf{m} : K^m \mapsto V^m$ used in Definition 6 to provide meta-information for each implementation is still unclear. It is kept deliberately loosely specified in terms of type to allow for a wide variety of research and application scenarios. It's closest equivalent in the compiler are *annotations*, which allow arbitrary INSPIRE nodes to be enriched with additional information. For the Insieme-RS backend, the information in INSPIRE node annotations is one important source of meta-information – the other is analysis performed on the INSPIRE sub-graph of each work item implementation version before or during code generation.

A powerful feature shared by both work item meta-information and INSPIRE annotations is that the data they represent can be richly typed. This includes not just support for basic types such as integers or floating point values, but also composite types like structures or lists, and, crucially, function types. This means that the compiler can internally generate a function modeling the behaviour of a work item implementation depending on any number of variables which need not be available at compile-time, and forward this function as work item meta-information to Insieme-RS. The runtime system can then use its information advantage – it knows the exact state of the program and its input – to evaluate the function and use it to steer its behaviour.

Examples of possible meta-information stored in work items include:

- Feature vectors used in machine-learning optimizations during runtime. These could e.g. contain the number of vector operations within the code region, or the ratio of memory accesses to arithmetic operations.

- Functions estimating some non-functional parameter of the work item. E.g. a function estimating its execution time or memory usage based on a set of parameters known at runtime and a model constructed by the compiler.

- Optimization goals supplied in the input program. E.g. in a real-time application, a work item representing a decoding step may carry meta-information indicating that it needs to complete in a certain amount of time, but not faster. Insieme-RS can then tune its execution to use the least amount of resources while still fulfilling the indicated performance requirement.

Chapter 5 demonstrates the real-world applicability and impact of this approach.

3.5 Implementation Notes

In this section some aspects of the implementation of Insieme-RS will be presented with more technical detail. Giving a full technical description of the entire runtime system at this level of detail would be infeasible in terms of volume, and also often simply mirror the state of the art – as with any large software project. Therefore, we will focus on those aspects of Insieme-RS which are either unique to the system, or implemented in a novel way which is made possible by the specific requirements of Insieme-RS.

3.5.1 General Overview

Insieme-RS is implemented in C99 [41], with a small number of components written in inline assembler for select platforms to increase performance. The primary component this type of optimization was performed for is the user-level context switching functionality, which is used to implement work item scheduling on workers.

The runtime system is implemented as a set of headers only. This has the advantage of enabling the back-end compiler to more easily perform full optimization and the inlining of small runtime functions. Additionally, it ensures that each stand-alone program produced by the compiler can be run on any supported system, without the need to provide separate stand-alone libraries. The main disadvantage of this approach is the fact that the whole runtime system has to be compiled every time an application is compiled. However, since it is pure C, the time for this is insignificant on most modern computers.

Figure 3.6 illustrates a basic overview of the components comprising Insieme-RS. Its components are grouped in three major categories:

Abstraction Components that abstract facilities provided from the OS level to enable portability. This is the only component in which external functions and interfaces which are not part of C99 or its standard libraries are used.

Core Components providing the core functionality of the runtime system.

Utilities Modules which offer functionality required by, but independent from the core components.

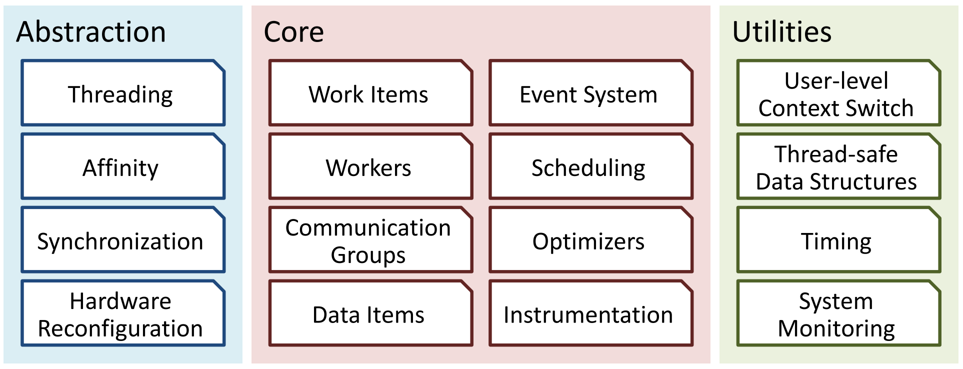


Figure 3.6: Insieme-RS implementation component overview.

The *abstraction* layer is intended to encapsulate basic OS-level threading, synchronisation, and other tasks which require a system-specific implementation (e.g. affinity settings) and provide implementations for each supported platform. At the point of writing, it features a POSIX [58] implementation targeting Linux and all compatible systems, as well as a Windows implementation which is based on native Win32 threading and synchronization libraries.

Utilities are components that are largely independent of the rest of the runtime system, but are used by it to accomplish its tasks. Currently, the following utilities are implemented:

User-level Context Switching A user-level (lightweight) threading and context switching library which provides high-performance implementations for x86 and x86-64 architectures and features a thread switching overhead of less than 10 cycles. A fallback implementation using the POSIX `ucontext` library is also provided for unsupported hardware.

Thread-safe Containers High-performance, thread-safe container structures based on atomic operations and fine-grained locking. Includes stacks, deques, counting deques, lookup tables and circular buffers. Some of these are described in more detail in Section 3.5.3.

Timing Highly accurate timers and sleeping functions, as well as calibration support to convert to/from CPU ticks and time units.

System Monitoring Provides information about the system state, such as the current externally generated CPU load.

Finally, *core* components encompass the central implementation of the runtime, including all the program objects and system-level components detailed in Sections 2.3 and 3.3.

Work Items The basic unit of work in the runtime system, as formalized in Definition 8. A user-level thread with additional meta-information about its data requirements and properties, with any number of implementations.

Workers OS-level threads which accomplish the scheduling and management of work items, see Definition 11.

Communication Groups Groups of work items which can perform aggregated operations, as defined in Section 2.3.4. Includes barriers, work distribution methods and aggregated data exchange functions operate on work groups.

Data Items The basic unit of data in the runtime, see Definition 9. Programs use data items to enable the runtime to optimize the distribution and usage of data throughout a program.

Event System A generic system that allows the components of the runtime – work items, workers, groups and data items – to generate and receive events. Events, formally introduced in Section 3.3.2, consist of a source, which is one of the above objects, an event ID and an event lambda, which is a combination of data and function executed when the event occurs.

Scheduling This component includes two separate systems: task scheduling and loop scheduling. Task scheduling implements the distribution of work items to workers, by means such as task stealing. Loop scheduling distributes the iterations of a loop among the members of a work group. Both are described in further detail in Section 3.3.3.

Optimizers The optimization of various processes performed by components listed above, such as scheduling and data distribution, is relegated to the optimizers in this module.

Instrumentation Instrumentation provides information about the non-functional behaviour of programs executed by the runtime as well as the runtime itself, either during execution or post-mortem.

3.5.2 Compiler → Runtime Mapping

Section 3.4 Provided a high-level description of the mapping of various parallel structures in OpenMP to the Insieme intermediate representation, and how that representation is in turn mapped to the structures of the runtime system. In this section, more detail on how work items descriptions and data items are implemented in the program generated by the Insieme-RS backend of the compiler will be provided.

The backend collects all work items (implicitly) defined within the program, including any entry points and regions within a parallel spawn construct. For each description of such a work item, a line in the *work item description table* is generated. Similarly, the *type table* contains an entry for each data item type used in the program. Both of these structures are gathered within a *program context*, which is initialized by a function generated by the compiler backend.

```
typedef struct _irt_context {
  irt_context_id id;
  irt_client_app* client_app;
  uint32 type_table_size;
  irt_type* type_table;
  uint32 wi_desc_table_size;
  irt_wi_description* wi_desc_table;
} irt_context;
```

Listing 3.1: Program Context Data Structure

Listing 3.1 describes the content of a program context data structure. In this and all further code listings within this section, C99 syntax is assumed, and unimportant implementation details (such as forward declarations) may be omitted for the sake of clarity. Each `irt_context` contains a unique identifier, a pointer to a data structure identifying the client application, and the essential work and data item tables which will now be described in detail.

Work Item Description Table

The work item table is a statically allocated array of data structures filled by the compiler backend. Each entry in the array corresponds to one work item description $\overline{w}$, and the size of the array is equal ti the total number of work item descriptions in the program $|\overline{W}|$.

```
typedef struct _irt_wi_description {
  uint32 num_variants;
  irt_wi_implementation* variants;
} irt_wi_description;

typedef enum _irt_wi_implementation_type {
  IRT_WI_IMPL_SHARED_MEM, IRT_WI_IMPL_DISTRIBUTED,
      IRT_WI_IMPL_ACCELERATOR
} irt_wi_implementation_type;

typedef struct _irt_wi_di_requirement {
  irt_data_item_id di_id;
  irt_data_range range;
} irt_wi_di_requirement;

typedef void wi_implementation_func(irt_work_item*);
typedef void wi_di_req_func(irt_work_item*,
    irt_wi_di_requirement*);
typedef void wi_channel_req_func(irt_work_item*, irt_channel*)
    ;

typedef struct _irt_wi_implementation {
  wi_implementation_func* implementation;
  irt_wi_implementation_type type;
  uint32 num_required_data_items;
  wi_di_req_func* data_requirements;
  uint32 num_required_channels;
  wi_channel_req_func* channel_requirements;
  irt_wi_implementation_variant_features features;
  irt_wi_implementation_runtime_data rt_data;
} irt_wi_implementation;
```

Listing 3.2: Work Item Descritpion Table Data Structures

Listing 3.2 contains definitions for all the essential data structures and functions required to define a work item description. Every individual `irt_wi_description` consists of a fixed number of implementations of type `irt_wi_implementation`. Each implementation in turn comprises the following components:

- The `implementation` function pointer. This function actually implements the program semantics of single-entry single-exit region encapsulated by this work item implementation.
- An `irt_wi_implementation_type`, describing whether this implementation variant is intended to run on a single shared memory node, a cluster of nodes or an accelerator.
- The number of data items required by this implementation, and a pointer to a function which generates a set of `irt_wi_di_requirements`, each representing a sub-range of a particular data item.
- An equivalent setup for the communication channels required by the data item. The implementation type, data item and channel requirement functions together implement the resource requirement function q described in Definition 2.4.
- A `features` data structure of type `irt_wi_implementation_variant_features`. This structure can be used to transport arbitrary data or functions from the compiler to the runtime, and is used to attach meta-data to each work item implementation version as described in Section 3.4.3.
- Finally, a `rt_data` structure, which is not filled in by the compiler but rather used by Insieme-RS optimizers to store information associated

with each work item implementation (e.g. performance information about a work item representing a parallel loop, to be used for scheduling should the loop be encountered again).

Data Item Type Table

While data items as described in Defintion 9 are generated dynamically during program execution by means of the operations presented in Section 2.3.3, the types τ for each data item need to be communicated from the compiler to the runtime system. The data item type table accomplishes this task, and its central data structures are presented in Listing 3.3.

```
typedef enum _irt_type_kind {
  IRT_T_BOOL,
  IRT_T_INT8, IRT_T_INT16, IRT_T_INT32, IRT_T_INT64,
  IRT_T_UINT8, IRT_T_UINT16, IRT_T_UINT32, IRT_T_UINT64,
  IRT_T_REAL16, IRT_T_REAL32, IRT_T_REAL64,
  IRT_T_STRUCT = 0xFF00,    // complex type start
  IRT_T_UNION, IRT_T_FUNC, IRT_T_POINTER,
  IRT_T_ARRAY, IRT_T_VECTOR, IRT_T_CHANNEL,
  IRT_T_BASIC
} irt_type_kind;

typedef struct _irt_type {
  irt_type_kind kind;
  uint32 bytes;
  uint32 num_components; // 0 for basic types
  irt_type_id *components; // num_components entries
} irt_type;
```

Listing 3.3: Data Item Type Table Data Structures

Each entry in the type table corresponds to one `irt_type`, comprising the following components:

- A `kind`, either of basic type such as bool or of a complex, potentially composed type such as a struct.
- The number of `bytes` the type occupies in memory.
- A number of components if the type is composed of sub-types, which defaults to 0 for basic types.
- A list of the components of the type, which are in turn references to entries in the type table.

The size of each type in bytes is essential to the functionality of the runtime system, as it is used when allocating memory for data items composed of the type. The further specification of the sub-components of the types allows for reflection on the type at runtime and may e.g. be used to convert between little-endian and big-endian systems when transferring data items in a heterogeneous cluster.

3.5.3 Thread-Safe Data Structures

In the implementation of many components of Insieme-RS, *thread-safe data structures* are required. While any data structure can be made thread-safe by the application of coarse-grained locking, such an implementation would lead to performance and scalability issues, particularly in highly congested data structures. Lock-free data structures [1] would appear to lend themselves well to this purpose, but are currently only mature and available for a limited set of container types, such as stacks and queues.

There are two central data structures in the runtime system which require concurrent parallel access and are not suitable for a lock-free implementation. These data structures are *event tables* (used to implement the event system described in Section 3.3.2) and *circular work buffers*, which serve as the containers used to represent the work item queue q and pool p as per Definition 11.

Event Tables

The event system detailed in Section 3.3.2 is a core component of Insieme-RS, and used in the implementation of many basic functions. Events are defined to originate from one source object y, and each event has a event type t (see Definition 12). Multiple event handlers h can be registered for each such combination of source and event type. In programming terms, each such handler is a closure [79] – a combination of a function and its data environment.

Providing this functionality in a C program in a way which is both fast and thread-safe requires some infrastructure. In Insieme-RS, the event system is based on *event tables*. For each type of source object (e.g. work items or data items) a hash table of *event registers* is allocated. Hashing is performed on the object ID of the event source y, and collisions are resolved using separate chaining with list heads. This allows for parallelism using fine-grained locking on each line in the hash table.

Each event register contains an array of occurrence counts and an array of linked lists of event handlers, both of these arrays are equal in length to the number of event types t which exist for the given source object type (as listed in Table 3.1). Finally, each event handler consists of a handler function matching the semantics defined in Section 3.3.2 and an arbitrary data environment.

```
typedef bool (irt_wi_event_lambda_func)(irt_wi_event_register*
      source_event_register, void *user_data);

typedef enum _irt_wi_event_code {
  IRT_WI_EV_STARTED,
  IRT_WI_EV_SUSPENDED,
  IRT_WI_EV_RESUMED,
  IRT_WI_EV_FINISHED,
  IRT_WI_CHILDREN_FINISHED,
  IRT_WI_EV_NUM // sentinel
} irt_wi_event_code;

typedef struct _irt_wi_event_handler {
  irt_wi_event_lambda_func *func;
  void *data;
  struct _irt_wi_event_handler *next;
} irt_wi_event_handler;

typedef struct _irt_wi_event_register {
  pthread_spinlock_t lock;
  irt_wi_event_register_id id;
  uint32 occurrence_count[IRT_WI_EV_NUM];
  irt_wi_event_handler *handler[IRT_WI_EV_NUM];
  struct _irt_wi_event_register *lookup_table_next;
} irt_wi_event_register;
```

Listing 3.4: Event System Data Structures

Listing 3.4 contains the C definitions of the central data types used for work item event handling in Insieme-RS. Note that this code is a simplified representation specialized for work items – the actual implementation is generated from a parameterized macro representation for different source object types.

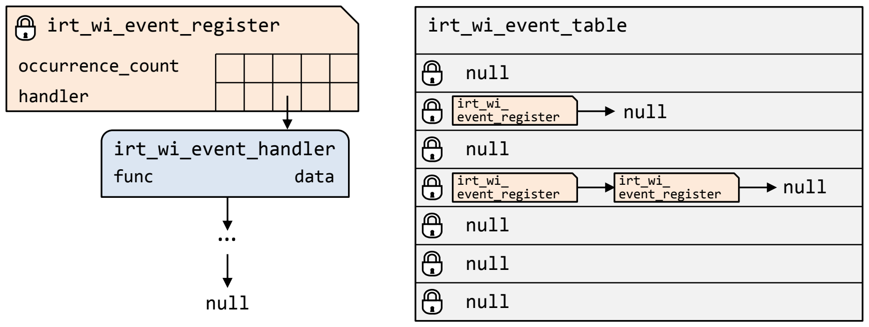


Figure 3.7: Insieme-RS implementation of event tables.

Figure 3.7 illustrates the relationship between the data structures involved in work item event handling, and the granularity of locking. While a full line of the hash table needs to be locked to find or add an event register, this lock can be released and only the register itself is locked to evaluate or manipulate event handlers. This distributed, fine-grained locking design allows for high event throughput even with a large number of parallel event invocations.

Circular Work Buffers

Workers store work items which are ready to execute in a queue and pool, as defined in Section 3.3.1. The queue and pool containers differ in semantics, but can be based on the same implementation. In Insieme-RS, multiple different versions of these essential container structures have been implemented. All of them need to support the following features:

- Store a sorted list of work items, which may be empty.
- Work items can be added and removed arbitrarily at the front (top) and the back (bottom) of the list, but not in its middle. We will call these operations *pushFront* and *pushBack* for adding elements and *popFront* and *popBack* for removing them.
- Any combination of these operations can be invoked in parallel.
- Determining the number of items in the container needs to be a low-cost operation (this is an important parameter for some scheduling algorithms, in which it is repeatedly queried).

A basic implementation would be a simple doubly linked list with global locking and a separate element count. However, such an implementation results in congestion whenever any two threads need to access any part of the container, even if they e.g. try to perform pushFront and pushBack operations which should semantically be able to execute in parallel.

While relatively mature lock-free implementations for queues exist, lock-free deques of arbitrary length which fulfill all the requirements outlined above are still a research topic and often quite complex [78]. For our purposes, a good balance between the sequential complexity of adding, removing and counting elements and parallel congestion is essential.

Circular work buffers in Insieme-RS were designed to fulfill the requirements outlined above. They are based on a fixed-size array of a length of 2^N which is indexed using 4 16-bit values packed into a 64-bit quadword. This indexing data structure has the advantage that it can be updated atomically on all target hardware platforms. Listing 3.5 depicts this data structure. The four indexing values are `top_val`, `bot_val`, `top_update` and `bot_update`.

```
#define IRT_CWBUFFER_LENGTH (1<<N)
#define IRT_CWBUFFER_MASK (IRT_CWBUFFER_LENGTH-1)

typedef union _irt_cwb_state {
  uint64 all;
  struct {
    union {
      uint32 top;
      struct {
        uint16 top_val;
        uint16 top_update;
      };
    };
    union {
      uint32 bot;
      struct {
        uint16 bot_val;
        uint16 bot_update;
      };
    };
  };
} irt_cwb_state;

typedef struct _irt_circular_work_buffer {
  volatile irt_cwb_state state;
  irt_work_item* items[IRT_CWBUFFER_LENGTH];
} irt_circular_work_buffer;
```

Listing 3.5: Circular Work Buffer Data Structures

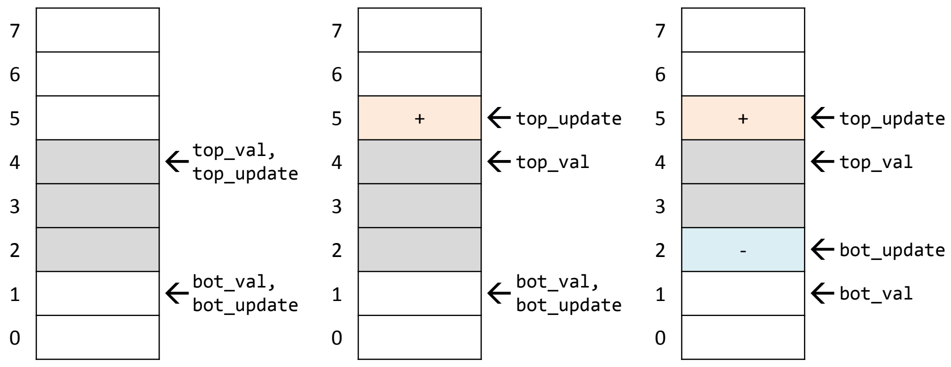


Figure 3.8: Insieme-RS circular work buffer semantics.

Figure 3.8 illustrates how circular work buffers are organised and how synchronisation is handled. The semantics of the indexing values are defined as follows:

- `top_val` (T^V) is the index of the work item at the front/top of the buffer.
- `bot_val` (B^V) is the index *before* the work item in the back/bottom of the buffer.
- `top_update` (T^U) is the same as T^V in the steady state of the container. During operations on the top of the buffer, it differs from T^V: if it is less than T^V, the elements between the two indices are being removed from the top of the buffer – otherwise, the elements between the two indices are being added.

- `bot_update` (B^U) is the same as B^V in the steady state of the container. During operations at the bottom of the buffer, it differs from B^V: if it is less than B^V, the elements between the two indices are being added at the bottom of the buffer – otherwise, the elements between the two indices are being removed.

Note that – due to the circular nature of the buffers – in this description, "less than" refers to the modulo space of size `IRT_CWBUFFER_LENGTH` in the following manner: T^U is considered *less than* T^V iff $(T^U < T^V \vee (T^U > T^V \wedge T^V < B^V))$. The second part of the condition covers the case where the buffer spans across the end of the array. Equivalently, B^U is considered *less than* B^V iff $(B^U < B^V \vee (B^U > B^V \wedge B^U > T^V))$.

Following these definitions, the 8-element buffer in Figure 3.8 is in these configurations, from left to right:

- Left: Steady state, no operation is being performed. Three work items are stored in the buffer, at indices 2, 3 and 4.

- Middle: A work item is in the process of being added on top of the buffer, at index 5.

- Right: While the addition of one work item at index 5 is still in progress, one work item is removed from the bottom of the buffer in an independent parallel operation.

Using this data structure and atomic operations on the state variable, parallel operations can be independently executed at the top and bottom of the buffer without any congestion, as long as the buffer is not empty. If multiple operation on the same end of the queue are invoked in parallel, one of them will succeed and the others will retry. Determining the number of elements in the container is trivial:

```c
uint32 irt_cwb_size(irt_circular_work_buffer* wb) {
  return (wb->state.top_val - wb->state.bot_val) &
      IRT_CWBUFFER_MASK;
}
```

3.5.4 Memory Reuse

Despite the advances made in generic memory allocator performance, particularly in multi-threaded scenarios [56], memory management is still a major performance hot spot in any runtime system. For all data structures with a potentially high frequency of allocations and deallocations (such as work items or event registers) Insieme-RS manages an individual reuse stack on each worker. When data structures are no longer in use, they are attached to their respective reuse stack on the current worker. When a new data structure of some type is required on a worker, it first checks its reuse stack for unused data structures of this type – new memory is allocated only if the reuse stack is empty.

Keeping the reuse stack local to workers has multiple advantages. It is cache- and NUMA-friendly, as data is only reused locally. Even more critically, it allows the omission of any kind of synchronisation or mutual exclusion for operations on the reuse data structures. Each worker only accesses its own reuse stacks, and consequently there is no possibility of any race conditions occurring.

However, in one use case this type of object-based local reuse has proven to be insufficient. When recursive, task-based programs with a high number of interdependent tasks are executed, the number of active stack spaces in memory required for execution can get very high. To solve this issue Insieme-RS uses *Asteroidea stacks*.

Asteroidea Stacks

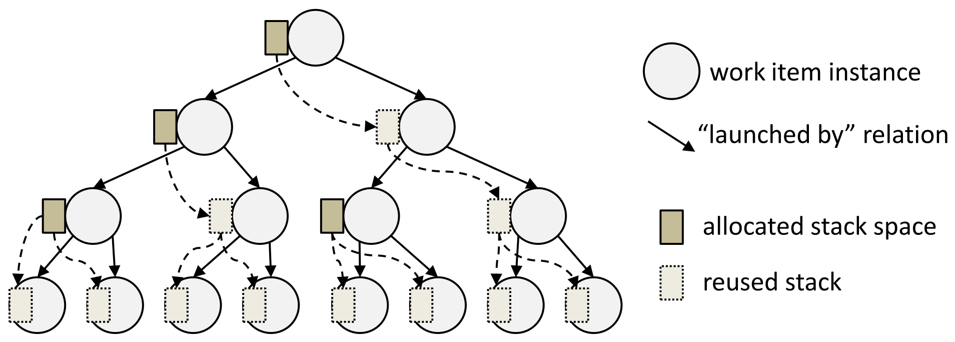


Figure 3.9: Reuse Pattern of Asteroidea Stacks.

Asteroidea stacks, named after the family of starfish for their ability to split up and function independently, are a mechanism designed to allow the *partial* reuse of stacks allocated for parent work items by their children. They are based on the observation that, in almost all task-based recursively parallel programs – which are usually the type of programs generating large amounts of work items – a work item will first create some arbitrary number of child work items, and subsequently wait for their completion by invoking the `join_all` operation (see Section 2.3.2). As the parent work item can only continue its execution after all its children have completed, these children are free to reuse any remaining stack space available on the parent stack.

To implement this behavior, any time a work item is started, Insieme-RS checks whether it has a parent, and if so, whether this parent is currently suspended in a `join_all` operation. Finally, an additional flag governing the availability of the parent work item's stack is checked. If it can be atomically acquired, this stack is reused for the child work item. This process can be repeated arbitrarily for further children of the child work item.

Figure 3.9 illustrates how stacks could be reused in a recursively task-parallel program where the launched-by relation L forms a binary tree (e.g. a basic task-recursive implementation of the Fibonacci algorithm). Note that this is only one possibility, and the exact reuse pattern depends on the number of workers executing the program as well as the scheduling policy employed.

	24 workers		4 workers	
N	S (MB)	A (MB)	S (MB)	A (MB)
5	232	216	96	56
10	552	288	400	144
20	1680	296	1520	144
30	2800	296	2640	144
40	3920	296	3864	144

Table 3.3: Asteroidea Stack Impact on Memory Use.

Table 3.3 lists stack memory usage measurements for a program forming a binary tree of work items such as the one illustrated in Figure 3.9. N represents the height of the tree, the S columns contain the memory use (in MB) without Asteroidea stacks, while they are active for the measurements in the A columns. With Asteroidea stacks, the maximum memory usage is a function of only the number of parallel workers used, while it grows with N for the default case.

Chapter 4

Topology-aware Multi-Process Scheduling

4.1 Introduction

Due to recent developments in hardware manufacturing, the number of cores in shared memory systems is rising sharply. Nowadays it is not unusual to find 32 or more cores in a single multi-socket multi-core system, possibly with an even larger number of hardware threads. The topology of these systems is often complex, with hierarchies comprising multiple levels of cache and heterogeneous access latencies in a non-uniform memory architecture (NUMA). Future many-core architectures [70] are likely to further increase the architectural complexity. While OpenMP [16] is one of the most widely used languages for programming shared memory systems, particularly in the field of High Performance Computing (HPC) [33], existing methods and implementations are often not well suited for this evolving hardware landscape.

This chapter presents experiments demonstrating that many existing OpenMP applications and implementations fail to scale fully on such shared-memory, multiprocess systems (SMMPs). They also do not take into account modern CPU power saving technologies increasingly employed in the name of green computing, which usually work on a per-socket basis [38]. As one possible way to overcome these difficulties and enhance throughput we propose the *centralized process-level scheduling of multiple OpenMP workloads* (*jobs*), taking into account available topology information. In this context we define *throughput* as the number of OpenMP jobs completed in a given time period. Our approach is immediately applicable to existing applications without any changes required from the user or programmer, which is a significant advantage considering the large number of OpenMP codes in active HPC use.

The contributions which will be presented in this chapter are as follows:

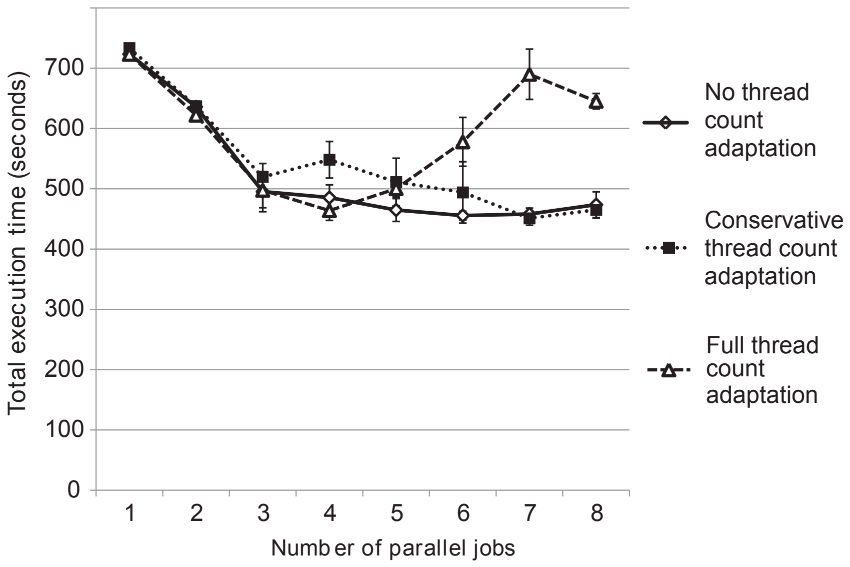
- A client/server architecture for centralized mapping of all OpenMP parallel workloads in a system to the available hardware resources.
- An implementation of this architecture as part of the Insieme compiler and runtime system [28].
- A topology-aware scheduling algorithm optimizing throughput and power consumption of SMMPs processing OpenMP workloads.
- Evaluation and analysis of the actual performance of our architecture and scheduling algorithm in terms of both execution time and power consumption. We compare our results to the Insieme compiler without multi-process management of parallel jobs as well as to results obtained using GCC's GOMP OpenMP library [60].

The remainder of this chapter is structured as follows: In the following section, we present benchmarks and analysis serving to explicate the problem and motivate our multi-process scheduling approach. Section 4.3 gathers some references to related work. Section 4.4 describes the architecture as well as the implementation of our client/server OpenMP runtime system and scheduling algorithm. The results of simulations and experimental evaluation are gathered in Section 4.5. Finally Section 4.6 presents a conclusion, and an outlook on potential future improvements.

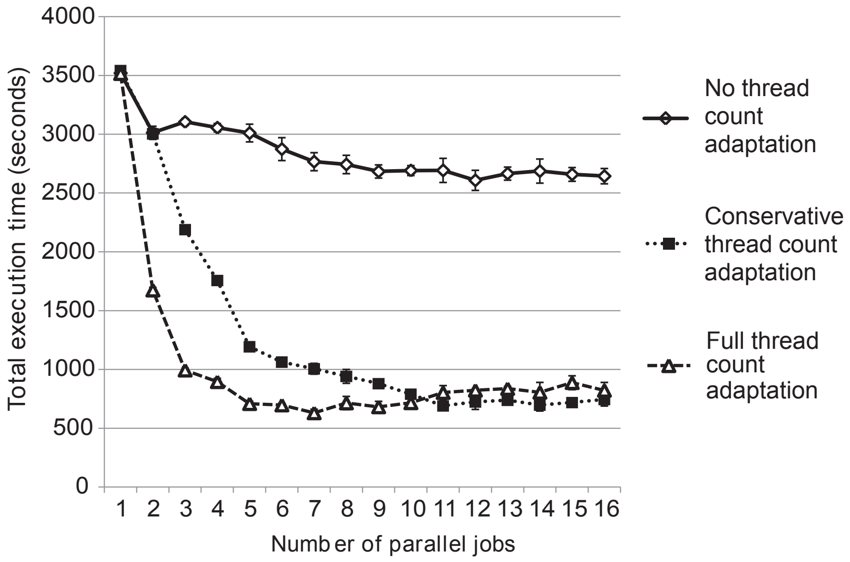
4.2 Motivation

We start our discussion by assuming a n-core SMMP system and m OpenMP programs (*jobs*) that should be executed on this system. Additional jobs can be added at any time. This situation corresponds to a realistic scenario in scientific computing and is the basic assumption for the experimental setting adopted throughout this chapter. There are a number of natural options for executing and scheduling multiple OpenMP jobs:

- Sequentially execute the jobs, and have each job allocate n threads – the default specified by the OpenMP standard. Standard queuing system in HPC clusters use this method.
- Start all m jobs in parallel and leave thread scheduling to the OS. As we will show, this option can have a severe detrimental impact on the resulting performance (see Section 4.5.3).
- Run less than m jobs in parallel, each of them using less than or equal n threads. A standard OpenMP implementation enables this option, which however requires some manual queuing. The thread scheduling is still left to the OS.



(a) Set of 8 jobs, large problem sizes.



(b) Set of 16 jobs, various problem sizes.

Figure 4.1: Multi-process scheduling: initial experiments.

Figure 4.1 shows some initial benchmarking results. For a complete description of the experimental setup and hardware used throughout this chapter see Section 4.5.3. The three parallel execution strategies shown relate to the options presented above as follows: *No thread count adaptation* simply uses n threads per job ignoring the number of parallel jobs running in the system; *Conservative thread count adaptation* uses $\min(n, \lfloor n/(0.5 * m) \rfloor)$ threads; *Full thread count adaptation* uses $\frac{n}{m}$ threads. With the first option, many more threads are running on the system than hardware cores are available, while for the third option the numbers are equal. The conservative option presents a compromise between these two strategies.

In both Figure 4.1(a) and 4.1(b), simple serial execution (1 parallel job) is clearly shown to be far from ideal. For the batch of 8 large jobs featured in the first experiment, an improvement in total runtime of 37% compared to serial can be achieved by exploiting job parallelism, while for the second batch an impressive 5-fold increase in throughput was measured. This underutilized performance potential is the motivation behind our approach of introducing a novel, job-level OpenMP scheduling.

4.2.1 Scaling behavior of popular OpenMP codes

In order to explain the performance improvements of job-parallel OpenMP execution demonstrated above we investigated the scalability of the OpenMP codes contained in the NAS parallel benchmarks [9] (BT, LU, MG, CG, IS, FT and EP) and two locally developed simple kernels (mmul and gauss, performing dense matrix multiplication and Gaussian elimination respectively). While our initial tests dealt with programs as monolithic units, we now determine the scalability of each OpenMP parallel region separately, since the differences between e.g. initialization code and actual computation or different phases of computation make a per-region analysis more informative and effective.

Figure 4.2 shows the scaling behavior of each parallel region contained in the test programs. Regions in the chart are identified by the program name, program size and line number of the first statement inside the region. As an example, `bt.B:bt130` identifies the parallel region starting at line 130 of the BT benchmark in the NAS suite, when run using the predetermined problem size B.

Note that, while this approach provides a relatively fine-grained view of the performance characteristics of the parallel regions, it does not account for the effects of external system load on region scalability. All test in this chapter were performed on a system without any applications running which are not managed by our infrastructure. Conversely, Chapter 5 presents an approach that adaptively adjusts parallel execution to a variable external system load.

In Figure 4.2, *optimal scaling limit* denotes the upper limit on the num-

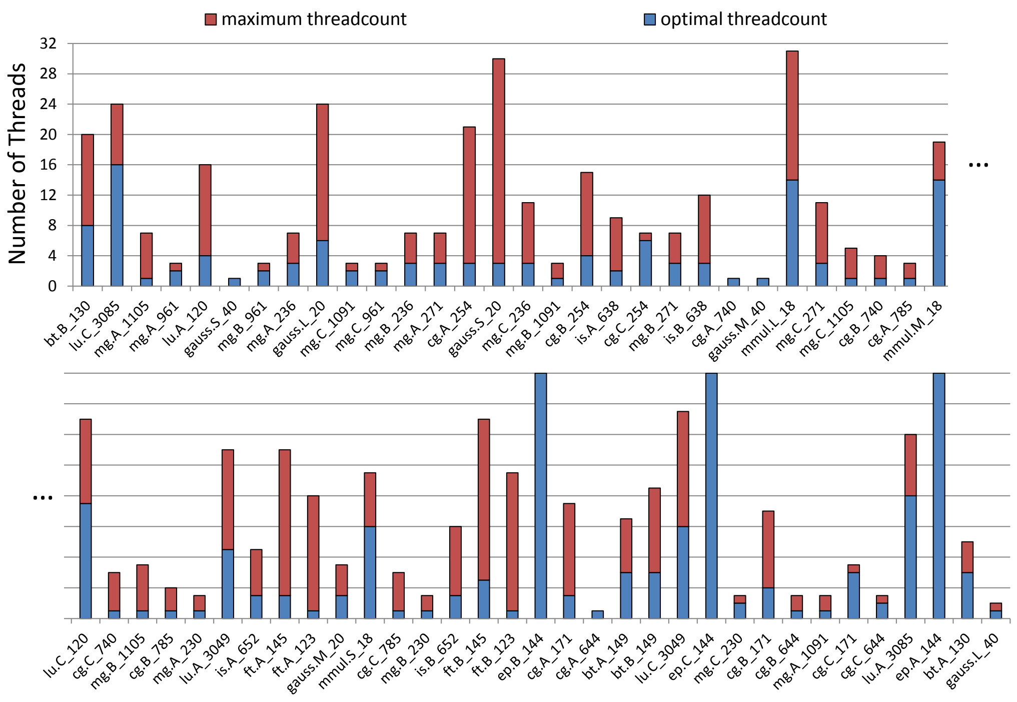


Figure 4.2: Per-region scaling of OpenMP codes. Regions identifiers are shown along the x-axis.

ber of threads at which the speedup obtained lies within 20% of ideal (as defined in Section 2.2.2). The *maximal scaling limit* is the number of threads with the largest absolute speedup. Using the latter achieves the lowest execution time for a single job, while the former is more power efficient and less wasteful if multiple jobs are to be run in parallel. Note that, for many parallel regions, the default OpenMP behaviour of choosing the same number of threads across all program regions as there are hardware cores available is greatly inefficient on our 32 core target system.

4.3 Related Work

Enhancing OpenMP to make better use of locality and increase scalability, particularly on multicore architectures, is a topic that has been repeatedly investigated over the past years [23, 59]. Recently, particular attention has been paid to scheduling tasks (as introduced by OpenMP 3) [30]. However, these efforts focus on improving the scalability and performance of single OpenMP programs. Conversely, our system overcomes individual program's scalability limitations by scheduling additional programs, optimizing the whole system's throughput. In the existing literature, locality is often signaled by nested parallel regions, which are still not widely used in practice. Our system simply improves locality of subsequent threads of each process, but does so for multiple processes at a time. A popular approach to developing scalable software for large machines is hybrid OpenMP/MPI programming [51]. Since this method produces multiple OpenMP processes, it is complementary to our process-level scheduling approach.

The scheduling of multiple independent processes is generally considered to be the duty of the operating system, not user-level software. However, one of the primary goals of multiprocessor scheduling at the OS level is fairness [54], while our OpenMP scheduling system is intended to optimize throughput. Even beyond that, the defining difference between OS-level scheduling and OpenMP process scheduling is that in the latter case the number of threads used can be determined by the scheduler itself, while it is fixed by user-level applications in the former. Also, when dealing only with OpenMP programs, some more useful assumptions can be made, like data locality being most important between subsequent threads (due to the default loop distribution strategies defined in OpenMP).

4.4 Method

Our topology-aware OpenMP job scheduling system consists of three major components:

- A component in the **Insieme compiler** which enables per-region profiling (e.g. for OpenMP parallel regions) and unique region identifiers. These can be implemented using meta-data on the work item descriptions corresponding to each OpenMP parallel region, as described in Section 3.4.3.

- **Insieme-RS** managing each client application. It is configured to work like a standard OpenMP library for most operations, but communicates with a central server when opening and closing a parallel region.

- The **Insieme OpenMP server**, a management process that keeps track of available CPU cores and outstanding requests for threads, and makes mapping and scheduling decisions based on system topology and load.

Note that the work presented in this chapter was completed using an early version if the Insieme runtime system. This required a separate server and client infrastructure. In its current version, Insieme-RS is capable of managing and executing work items from multiple programs within a single process, further reducing communication overheads.

4.4.1 Process Communication

Process communication is required to implement the centralized management of system resources across Insieme-RS processes. In practice this means that multiple Insieme-RS client applications – upon encountering a work item invocation corresponding to the start of a parallel region – request hardware resources from the management process (server), including information about the expected scalability of the related region. The server then decides which and how many cores to dedicate to the requesting process and sends a reply indicating them.

In our system this communication is achieved by means of UNIX System V message queues [75]. The message queue mechanism was chosen because of its good semantic fit with the desired functionality and relatively low overhead of less than 6 microseconds for each paired send/receive operation on our hardware.

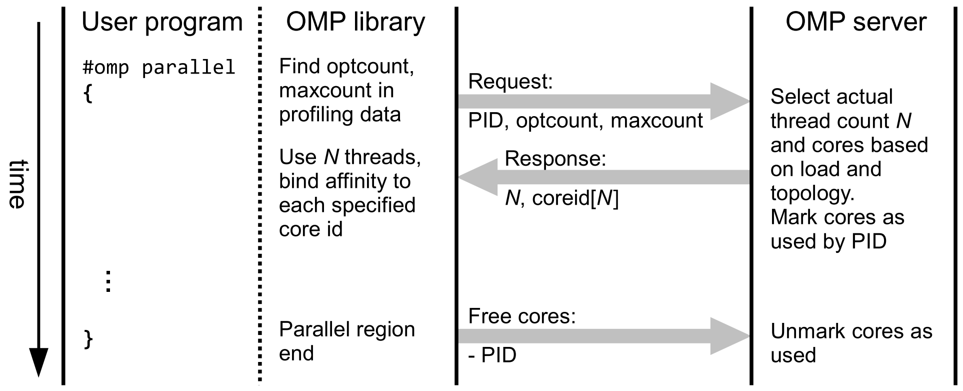

time
User program
OMP library
OMP server
#omp parallel
{
Find optcount, maxcount in profiling data
Request:
PID, optcount, maxcount
Select actual thread count N and cores based on load and topology. Mark cores as used by PID
Use N threads, bind affinity to each specified core id
Response:
N, coreid[N]
⋮
}
Parallel region end
Free cores:
- PID
Unmark cores as used

Figure 4.3: Process communication.

Figure 4.3 illustrates the typical communication operations associated with each parallel region in the source program. *PID* stands for the unique process id of the user program, *optcount* and *maxcount* refer to the scaling descriptors introduced in Section 4.2.1 and gray arrows represent communication over a message queue.

When a new parallel region is encountered (equivalent to the spawning of a work item), its unique identifier (generated at compile time and stored as meta-data with the work item description corresponding to the parallel region) is looked up in the table of available profiling information by the runtime system. The retrieved data is included in a request dispatched to the central server process, which additionally includes the process ID of the user program making the request. Upon receiving this request, the server makes its mapping and scheduling decisions (see next section for details). Unless the system is fully loaded (in which case the request is postponed) a reply is sent immediately, containing the number of threads/cores assigned

(N) as well as a list of core ids to be used. The runtime system on the client side then sets up the requested number of workers, and binds each of them to a specific core as specified by the list sent by the server. After dispatching the reply, the server flags the cores it indicated as in use in its internal data structures.

4.4.2 Topology-aware Scheduling

In multi-socket multi-core NUMA systems, there are multiple levels of memory hierarchy to be aware of, with significant differences in terms of access latency and bandwidth. An example of such a hierarchy, ordered from fastest to slowest: L1 cache, L2 cache, shared L3 cache, node local RAM, RAM one hop distant, RAM n hops distant. We call a scheduling process *topology-aware* if it seeks to minimize memory accesses to slow, distant memories by explicitly making use of information on the structure of a system.

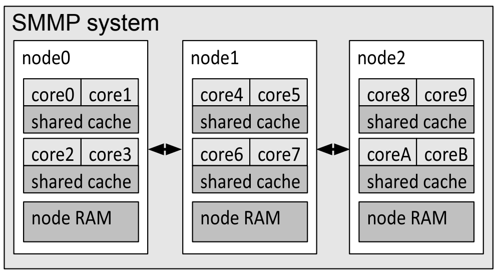
One critical difference between scheduling as it is generally encountered in e.g. OS-level schedulers and our OpenMP job scheduling is that, due to the flexibility of most OpenMP programs, we can freely decide not just which threads to run when and on which hardware, but also the number of threads to use for specific regions of a program. This decision should be based on knowledge about the scalability of the regions in question (currently gained through profiling) as well as the current and expected future load of the system.

For the implementation of our system we use information gained from `libnuma` [48] and the Linux `/proc/cpuinfo` mechanism to construct a distance matrix with one distance value for each pair of cores. From here on, we refer to the the entry at position i, j in this matrix as $\text{dist}(i, j)$. These numbers correspond to the total latency of all the connections in a path $(e_0^{\mathcal{H}}, e_1^{\mathcal{H}}, \ldots, e_n^{\mathcal{H}})$ between two CPUs $i = e_0^{\mathcal{H}}$ and $j = e_n^{\mathcal{H}}$ in a NUMA system according to the hardware model introduced in Section 2.1:

$$\text{dist}(i, j) = \sum_{k=0}^{n-1} l(e_k^{\mathcal{H}}, e_{k+1}^{\mathcal{H}})$$

In practice, it is often not possible to determine these values exactly, however, as long as the relative orders of magnitude between them are correct they are useful for topology-aware scheduling. In Section 4.5.1 we demonstrate that, on our evaluation system, the distance estimates obtained using [48] and the Linux `/proc/cpuinfo` mechanism are strongly correlated with real latency measurements taken by a microbenchmark.

Figure 4.4(a) provides an example topology of a relatively small system, and Table 4.4(b) shows the corresponding distance matrix. Note that the factor of distance amplification for each higher level of hierarchy can be chosen arbitrarily, but should always be larger than the maximum distance possible on the previous level. For clarity we have chosen a value of 10 in the example.



(a)

	0	1	2	3	4	5	6	7	8	9	A	B
0	0	0	1	1	10	10	10	10	20	20	20	20
1	0	0	1	1	10	10	10	10	20	20	20	20
2	1	1	0	0	10	10	10	10	20	20	20	20
3	1	1	0	0	10	10	10	10	20	20	20	20
4	10	10	10	10	0	0	1	1	10	10	10	10
5	10	10	10	10	0	0	1	1	10	10	10	10
6	10	10	10	10	1	1	0	0	10	10	10	10
7	10	10	10	10	1	1	0	0	10	10	10	10
8	20	20	20	20	10	10	10	10	0	0	1	1
9	20	20	20	20	10	10	10	10	0	0	1	1
A	20	20	20	20	10	10	10	10	1	1	0	0
B	20	20	20	20	10	10	10	10	1	1	0	0

(b)

Figure 4.4: Topological core distance example.

Cores are selected based on a greedy algorithm that locally minimizes the distance from the previously selected core to the next one. This is performed rapidly via lookups in statically cached lists of *close cores*. While not always resulting in a globally optimal core selection, the low overhead and importance of local distances for many algorithms (see Section 4.5.2) make this method well suited for our purpose.

Our OpenMP job server supports some flags to adjust the core selection and mapping process, which will be described next. *Fragmentation* in this context means that, over time, threads belonging to many different OpenMP processes will be assigned to cores belonging to a single topological unit (e.g. ones sharing a level in the memory hierarchy), due to earlier scheduling decisions. This has negative effects on locality and the effectiveness of shared caches.

Locality Turns on and off the use of topology information to improve locality. Useful to check the base assumption that higher locality improves performance.

Clustering Reduces fragmentation over longer running periods by preferentially maintaining clusters of close resources as free or occupied. This approach can also reduce power consumption on systems with per-node power management (see Section 4.5.3). However, it induces small overhead costs in processing time and server memory requirements.

Enhanced Clustering In conjunction with clustering, allows the server to further reduce fragmentation by slightly decreasing the number of cores provided to a process in cases where a new, previously unused set of cores would become fragmented when using an unmodified selection. Abbreviated as `[ehc]` in the algorithmic description.

Strict Thread Counts If enabled, the Insieme OpenMP server may postpone requests when highly loaded instead of starting them with a smaller number of threads than ideal. Beneficial when there is a mixture of jobs with varying scalability.

The impact of these options is examined in Section 4.5. Note that minimizing distance by means of clustered scheduling is not always ideal on a NUMA system which is not fully loaded, since memory bandwidth intensive applications may benefit from threads being spread out across nodes. However, our algorithm is optimized for the case of the full system being utilized by numerous threads from multiple processes.

Algorithm 4.1
Topology-aware multi-process scheduling core selection algorithm.

CA	number of cores available
RT	target number of cores
C, PC	core identifiers
LC	list of core identifiers
`optcount`, `maxcount`	request parameters (Sec. 4.4.1)
[`strict`], [`local`], [`clustering`], ...	scheduling flags (Sec. 4.4.2)
`loadfactor`, `threshold`	$\in [0, 1]$ depending on system state

```
$RT = \texttt{optcount} + \texttt{loadfactor} * (\texttt{maxcount} - \texttt{optcount})$
**if** $CA < 0$ **or** ([strict] **and** $CA <$ optcount) **then**
    put current request on FIFO queue
    **return** $\{\}$
**end if**
**while** $RT > 0$ **do**
    $C$ = Free core
    **if** [local] **then**
        Choose $C$ from close core list of $PC$
        **if** [clustering] **and** $C$ from new set **then**
            Prefer $C$ from (in order):
            $\hookrightarrow$ Occupied core set containing exactly $RT$ free cores
            $\hookrightarrow$ Occupied core set containing $> RT$ free cores
            $\hookrightarrow$ Any free core set
        **end if**
        **if** [ehc] **and** $\text{dist}(PC, C) > \texttt{threshold} * |\text{size}(LC) - RT|$ **then**
            **return** $LC$
        **end if**
        $LC = LC \cap \{C\}$
        $PC = C$
        $RT = RT - 1$
    **end if**
**end while**
**return** $LC$
```

Algorithm 4.1 provides a definition of the decision procedure performed by the server when receiving a new request for threads from a client process. The *close core list* is a list precomputed for each core that contains the core identifiers of all other CPU cores in the system, listed in order of topological distance. For clustering, a *core set* is a set of cores which share some level of memory hierarchy. Consequently there can be multiple levels of core sets, in which case the algorithm tries to find suitable cores starting from the lowest level of hierarchy. In the example shown in Figure 4.1 there are 2 levels of core sets, the first sharing cache (e.g. cores 0 and 1) and the second sharing memory segments (e.g. cores 0 to 3).

When a client sends a message signaling the end of a parallel region, the cores are marked as free and, if outstanding requests are in the FIFO queue, they are processed as described above.

4.5 Evaluation

In this section our topology-aware multi-process scheduling algorithm is evaluated, firstly by performing simulations and calculating some theoretical metrics and secondly by performing experiments and measuring runtimes and power usage. All experiments were performed on Sun X4600 M2 servers with AMD Opteron 8356 processors. This is an 8 socket architecture, with 4 cores each containing private L1 and L2 caches and sharing 2 MB of L3 cache. The sockets have a distance of one to three hops each [77]. Figure 4.5 depicts this architecture in the model introduces in Section 2.1.

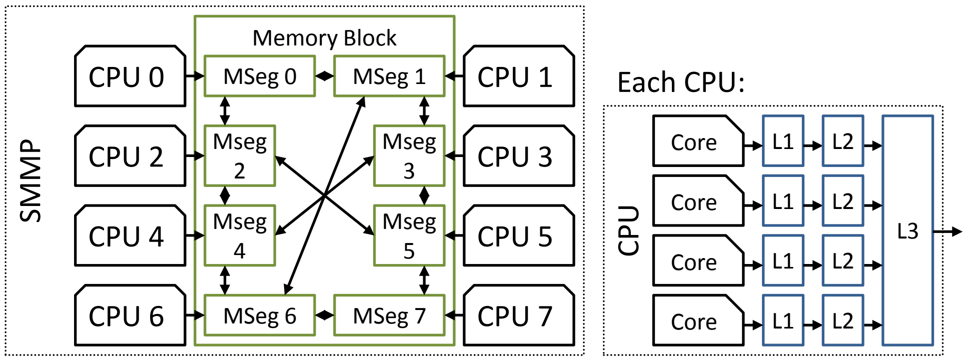


Figure 4.5: Evaluation system hardware architecture.

The systems run CentOS version 5 (kernel 2.6.18) 64 bits. To compile the reference version of the example programs, GCC version 4.3.3 was used with the -O3 option to reflect a production environment. As for our own software, SVN revision 277 of the Insieme source-to-source compiler and runtime system was employed, using the same GCC version and options as above to perform back-end compilation.

To ensure statistical significance each experiment was repeated 10 times, and the median result is reported. In charts vertical error bars are used to show the standard deviation of a set of experiments.

4.5.1 Topological Distance Matrix Verification

In order to verify our `libnuma`-based method for deriving a distance metric (described in Section 4.4) on our evaluation hardware, we have measured the latency of memory accesses between each pair of CPUs in the system. This is accomplished by creating a specifically formed array, which is designed to be iterated on using a series of indirect memory accesses. Each entry in the array is set such that the next address will be in a different cache line from the current one, as depicted in Figure 4.6 for a small example.

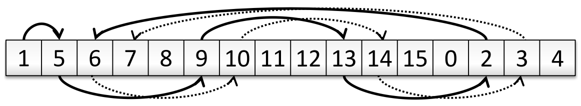


Figure 4.6: Memory latency indirection array (cache line size 4).

	0	1	2	3	4	5	6	7
0	579	664	626	897	739	735	801	944
1	665	572	897	641	727	718	690	780
2	673	747	498	735	618	618	780	780
3	780	664	727	498	611	605	780	769
4	780	763	611	619	498	732	690	780
5	780	753	619	611	718	489	748	664
6	780	671	727	747	641	897	579	664
7	944	780	739	734	897	626	672	572

	0	1	2	3	4	5	6	7
0	10	12	12	14	14	14	14	16
1	12	10	14	12	14	14	12	14
2	12	14	10	14	12	12	14	14
3	14	12	14	10	12	12	14	14
4	14	14	12	12	10	14	12	14
5	14	14	12	12	14	10	14	12
6	14	12	14	14	12	14	10	12
7	16	14	14	14	14	12	12	10

Figure 4.7: Distance matrices. Left: measured / Right: estimated

Figure 4.7 shows the distance matrices obtained on our evaluation system using this measurement method, as well as the distance provided by `libnuma`. The real values are depicted on the left, in nanoseconds. A value of 664 in line 0, column 1 means that it took 664 nanoseconds to perform an uncached memory access from a core belonging to cpu 0 to a memory location in memory segment 1.

These measured values correspond well to the theoretical values reported by `libnuma`, which are shown to the right. Crucially, for all CPUs, forming a *close core list* for Algorithm 4.1 based on the real, measured timings will be equally as valid as ordering them according to the theoretical results. Both matrices also conform to the expectations set by the system architecture illustrated in Figure 4.5 – the access latency and distance is a function of the number of hops between CPUs and their memory segments.

One curious, but perfectly repeatable, aspect of the measured data is the fact that access latencies are not completely bijective. For example, accessing memory in segment 3 from core 0 takes slightly longer than accessing memory in segment 0 from core 3. These results are believed to be related to the specific cache consistency implementation used in this particular hardware platform, and they do not significantly alter the relative order in each close core list.

4.5.2 Simulation

To evaluate the impact of the scheduling options presented in Section 4.4.2 we performed a simulation of client requests and calculated the following metrics:

Overhead The average amount of time required to make a scheduling decision, in microseconds.

Target miss rate The difference between the desired number of threads (RT) and the number actually provided (N).

Three distance metrics $LC = \{c_1, ..., c_N\}$ denoting the set of cores provided:

- Total distance: $\sum_{i=1}^{N} \sum_{j=1}^{N} dist(c_i, c_j)$
- Weighted distance: $\sum_{i=1}^{N} \sum_{j=1}^{N} \frac{dist(c_i, c_j)}{|i-j|+1}$
- Local distance: $\sum_{i=1}^{N-1} dist(c_i, c_{i+1})$

Which distance metric has the best predictive qualities depends on the access patterns the code exhibits. Weighted and local distance are more significant than total distance for many real-world OpenMP kernels which rely on the default distribution of loop iterations, resulting in data access locality being higher in subsequent (according to OpenMP numbering) threads.

			Distance		
	μs	MR	Total	Weight	Local
none	1.26	2.94	8003	2098	893
locality	1.28	2.94	6765	1537	485
clustering	1.3	2.94	6054	1289	340
clustering2	1.29	3.09	5760	1173	275
c2 + strict	1.3	2.07	7159	1340	327

Table 4.1: Multi-process scheduling metrics computed in simulation.

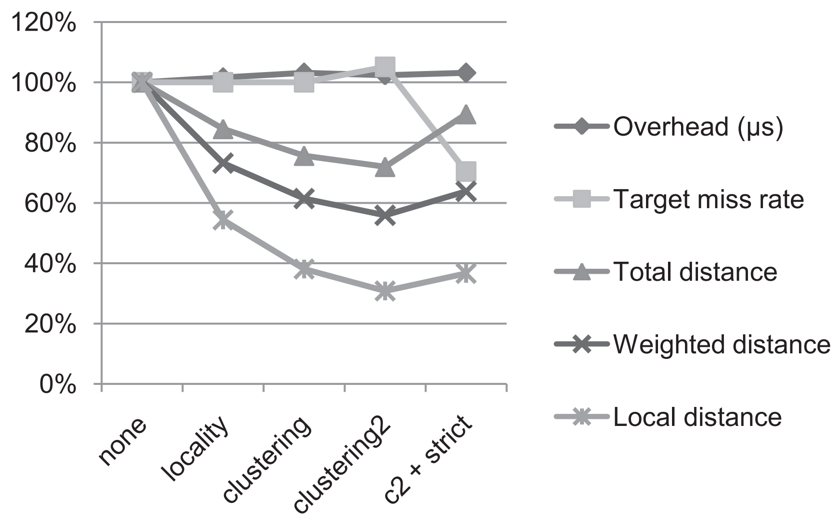


Figure 4.8: Relative multi-process scheduling performance metrics for different settings, computed in simulation.

In our simulation we evaluated the results of 1000 requests to the Insieme OpenMP server, with four different configurations corresponding to different settings of the flags introduced in Section 4.4.2: *none* disables all flags; *locality* enables the use of locality; *clustering* enables locality and clustering; *clustering2* enables locality, clustering and enhanced clustering; *c2 + strict* enables all flags. The simulation uses the same system topology as the experimental setup described in Section 4.5.3, randomly simulating jobs with sizes normally distributed and ranging from 1 to 32 threads.

Table 4.1 shows the raw values while Figure 4.8 illustrates the relative impact of the different scheduling options by normalizing the values. The columns contain, in order from left to right: the amount of time, in microseconds, required for the scheduling decision; the target miss rate; and the three distance metrics described above.

The impact on response time and target miss rate of the locality and clustering options is negligible, but they can reduce the weighted distance of the returned set of cores by around 40% and the local distance by up to 64%. Enabling the strict thread counts option significantly reduces the miss rate, but at the cost of reduced locality. In the next section the practical impact of these options will be evaluated.

4.5.3 Experiments

Figure 4.9 shows the results of a small-scale experiment performed to compare the total execution time of a fixed set of benchmarks using the following options:

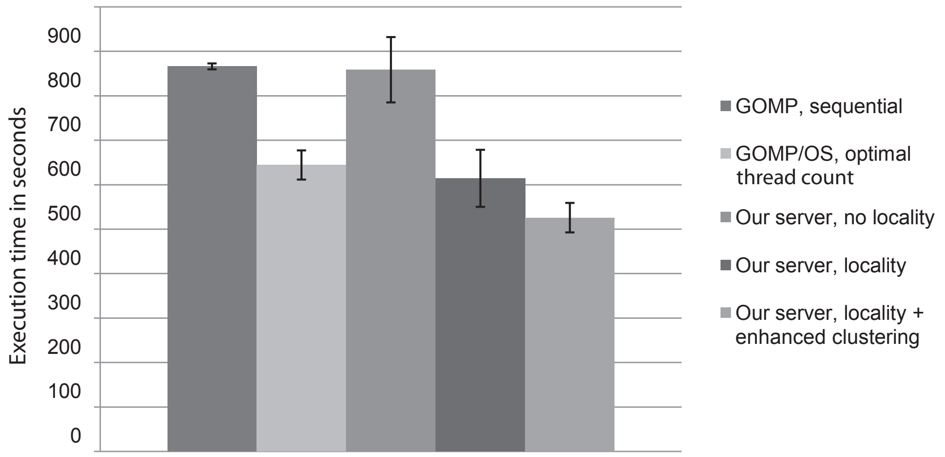


Figure 4.9: Multi-process scheduling – small-scale experiment.

GOMP, sequential : traditional sequential execution using the GCC compiler and GOMP OpenMP library.

GOMP/OS, optimal thread count : execution using operating system scheduling, the GCC compiler and GOMP OpenMP library and hard-coding the optimal number of threads for each benchmark (determined using an exhaustive search).

our server, no locality : Uses our multi-process scheduling system presented in this chapter, but without any topology information.

our server, locality : Our system with hardware topology information.

our server, locality + enhanced clustering : Our system with hardware topology information, with the *enhanced clustering* option presented in Section 4.4.2 enabled.

The specific set of programs used in this experiment was the following (randomly selected from the set of test applications introduced in Section 4.2.1): *mg.C, gauss.large, is.A, matrixmult.medium, cg.A, gauss.small, bt.B, mg.A, ep.B, ft.A, is.A, lu.A, matrixmult.small.*

The theoretical advantages of locality-based scheduling and clustering shown previously are confirmed by an improvement of 28% by exploiting the former and 39% by additionally enhancing the latter, compared to using our system without making use of topology information. The improvement compared to traditional sequential execution is 40%, and 19% remain when comparing standard OS parallel scheduling of the processes and statically forcing optimal thread counts. In other words, in this experiment, a reduction in total execution time of around 21% can be achieved by improving the exploitation of hardware parallelism by selectively using multiple parallel processes. On top of this improvement, an additional gain of 19% is possible by enhancing the locality of CPU sets executing parallel regions via topology-aware scheduling.

Large-scale Experiments and Power Consumption

As a second step of evaluation we performed a large scale experiment. Over 5 hours, a new OpenMP process was randomly selected and started every 10 to 60 seconds. The programs were again chosen from the set introduced in Section 4.2.1. The random number generator seeds for program and interval selection were kept constant throughout the experiment to guarantee repeatability and comparability.

During the runtime of the experiments, we continuously measured and logged the system power consumption using the Sun ILOM service [77], which allows for a resolution of several measurements per second.

Table 4.2: Multi-process scheduling – performance results of 5 hour experiment.

Scheduling type	Total Time (s)	% of sequential
Sequential	28356	100.00 %
OS parallel, no limit	121855	429.73 %
OS parallel, limit 8	82417	290.65 %
OS parallel, limit 2	23591	83.20 %
server, OS	21527	75.00 %
server, no locality	27959	98.60 %
server, locality	21240	74.91 %
server, clustering	18941	66.80 %

Table 4.2 lists the total runtime required to finish execution of all the programs launched during the 5 hour testing period, for various scheduling methods. *Sequential* refers to standard sequential execution, *OS parallel* executes a number (up to some limit) of processes in parallel without any explicit scheduling and *Server* uses our system. *Server, OS* only assigns the amount of threads to use, but leaves their placement up to the OS. The other options enable the corresponding flags described in Section 4.5.2.

Fully parallel execution using standard OS scheduling leads to a very large number of active threads and a performance collapse due to context switching overhead. While this method can achieve good results in the small-scale experiments shown in Section 4.2 care must be taken to select a suitable limit for production use. In this experiment, a limit of two parallel processes leads to an improvement of 17% compared to sequential execution.

Our client/server mapping system performs well, and is capable of greatly improving throughput compared to traditional sequential execution or nonmanaged parallel processes. With locality information and clustering the best result is achieved, a 33% improvement compared to sequential execution.

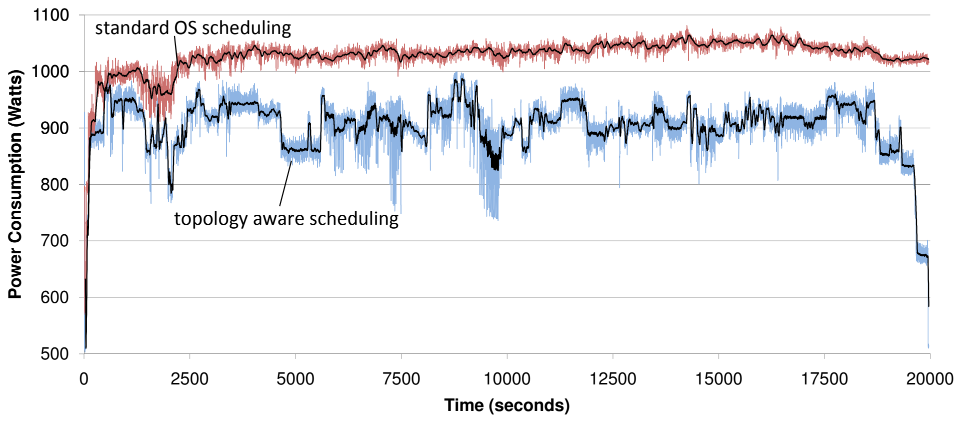


Figure 4.10: Power consumption over time during large-scale process scheduling experiment.

However, there is another important advantage to offered by clustered thread scheduling and affinity mapping. By reducing inter-node communication and preferentially keeping entire nodes and related core sets empty this technique allows hardware power saving technologies to function more effectively. On our target system, and most servers currently in use, power saving technologies like frequency scaling only work on a per-node basis and not on the individual cores of a node. Clustered, locality-aware scheduling preferentially keeps entire nodes free of work, allowing them to enter an appropriate low-power state.

Figure 4.10 shows the measured power consumption, over the time period of the experiment, of standard OS scheduling and clustered scheduling using our server. The average power consumption of the former is 1014 Watts, which the latter reduces by 12% to 904 Watts. Note that around 8 measurements per second are taken, and that the black lines represent a central moving average over 25 data points.

4.6 Summary

The process-level scheduling and mapping solution presented in this chapter uses system topology information to improve thread locality for multiple OpenMP programs executing in parallel. Additionally, a suitable number of threads is automatically selected for each OpenMP parallel region depending on scalability estimates and current system load. The implementation is based on the Insieme source-to-source compiler, a separate server process that manages hardware resources and the Insieme runtime system. Insieme-RS was adjusted to communicate with the server, managing the affinity of its workers as instructed to allow for cooperation between multiple processes.

The evaluation of various distance metrics in simulation indicates that the presented method succeeds in improving locality, and both small- and large-scale benchmarks show consistent improvements in processing throughput. Additionally, with clustering of related computations on various levels of the memory hierarchy, a marked decrease in average power consumption can be observed.

One drawback of our method is the need for reliable per-region scaling data. This necessitates either developer-supplied information or instrumented benchmarking runs. However, using the Insieme infrastructure this process can be largely automated.

Chapter 5

Automatic Loop Scheduling

5.1 Introduction

OpenMP [16] is one of the most widely used languages for programming shared memory systems, particularly in the field of High Performance Computing (HPC). Despite the introduction of task-based parallelism in recent versions of the standard [30], loop parallelism remains a very important part of most OpenMP programs. Thus, the question of how to map parallel loop iterations to threads and cores has been continually investigated since the standards' inception. In Section 5.3 we provide an overview of some of this existing work, and describe how our approach improves upon previous methods.

Like the entire Insieme-RS project, our loop scheduling system is built on the idea of *close integration between a state-of-the-art compiler providing in-depth analysis and a custom runtime library* that continuously monitors the overall system state while minimizing overhead. Such integration is realized by having the compiler generate a data structure for each parallel loop in the original program which captures analysis-derived meta-information about the loop body in addition to the actual executable code. This approach is immediately applicable to existing programs without any code-level changes, a significant advantage considering the large number of OpenMP codes in active HPC use.

We have implemented this system and evaluated its performance. Our contributions for automatic loop scheduling are as follows:

- A method using polyhedral model [14] based utilities to obtain effective estimates of OpenMP loop performance over all potential iteration ranges.

- A runtime loop scheduling algorithm that uses these estimators as well as current system state information to make loop scheduling decisions.

- An encoding of meta-information statically collected by the compiler into executable code usable during the runtime of a program.
- An implementation of this architecture in the Insieme compiler and runtime system [28].
- Evaluation and analysis of the measured performance of our scheduling algorithm in terms of program execution time. We compare our results to results obtained by the version of GOMP [60] included with GCC 4.5.3, using both its default scheduling policy and the best policy for each program determined by exhaustive search.

The remainder of this chapter is structured as follows: The next section will provide some experimental results that motivate our approach. Section 5.3 gathers some references to related work. In Section 5.4 we describe the architecture and implementation of our automatic loop scheduling method, including the compiler analysis, the runtime scheduling system and their interaction. The results of experimental evaluation are presented in Section 5.5. Finally Section 5.6 presents a summary, and an outlook on potential future improvements.

5.2 Motivation

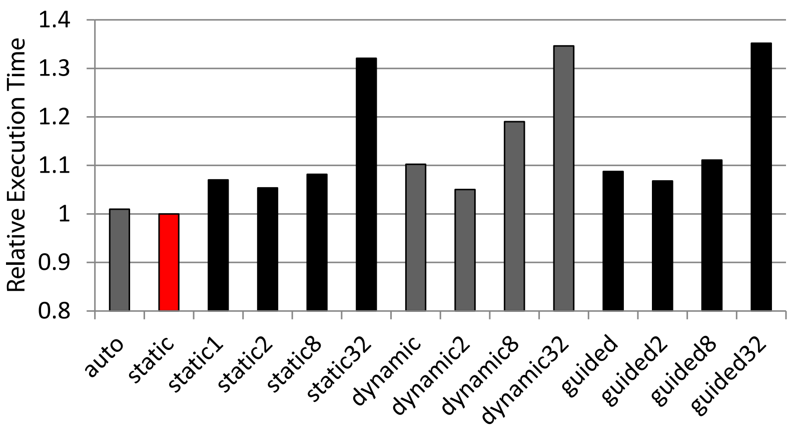
In this section we present some initial experiments using simple OpenMP kernels in a variety of settings. These results motivated our design of a unified compiler/runtime approach to loop scheduling. They also demonstrate the importance of load awareness. For a complete description of the experimental setup and hardware used throughout this chapter see Section 5.5. In all our figures the relative execution time normalized to the best performing configuration is shown.

We will investigate three separate factors – program characteristics, problem size and the degree of external load in the system – and demonstrate that all of these significantly influence the ideal choice of loop scheduling policy.

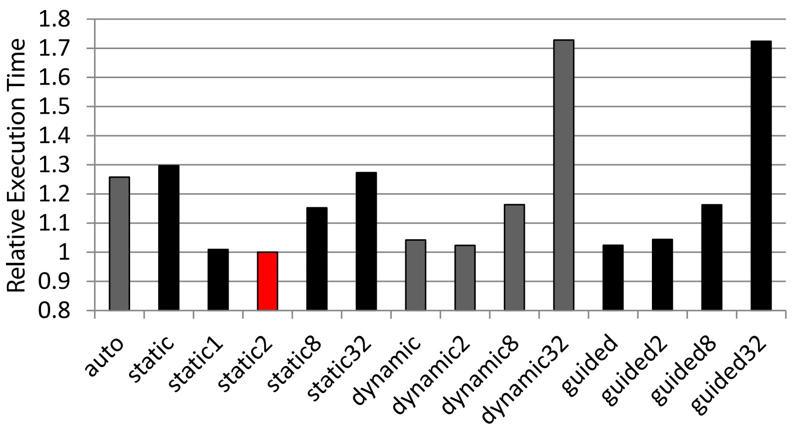
5.2.1 Impact of Program Characteristics

Figure 5.1 illustrates results for two kernels, dense matrix multiplication with full and triangular matrices, using a variety of standard OpenMP loop scheduling policies. Execution times in this chart, as well as all further figures in this section will be depicted relative to the best scheduling policy for the given program.

Clearly, the ideal loop schedule depends on the characteristics of the program. The dense matrix multiplication requires an equal amount of work



(a) Dense matrix multiplication

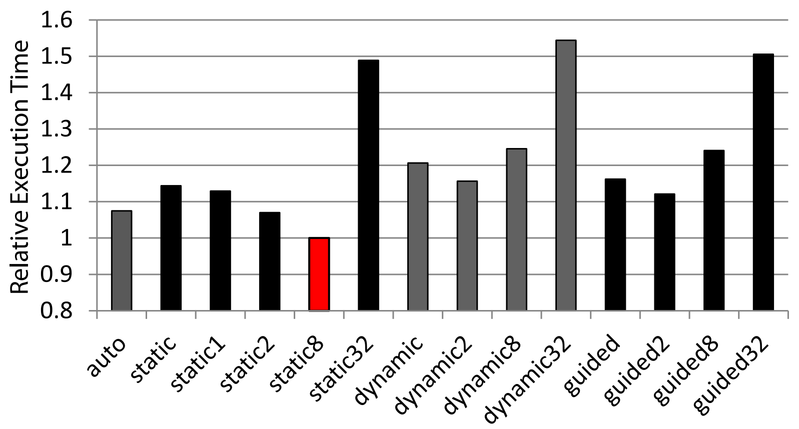


(b) Triangular matrix multiplication

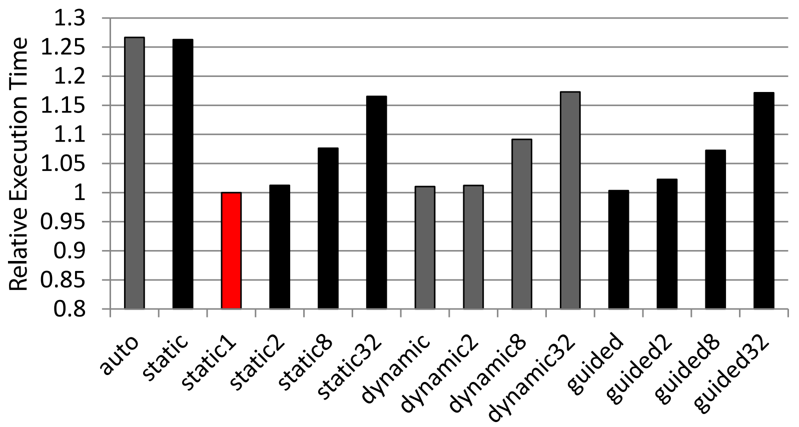
Figure 5.1: Initial experiments, impact of program characteristics.

within each iteration of the parallel loop while for the triangular matrix, the effort per iteration depends on the iterator value.

We say that the dense matrix multiplication has a *flat work profile* while the work profile for the triangular matrix is *slanted*. The former works well with fully static scheduling, because each equally sized chunk will perform an equal amount of work. In the latter case, a round-robin loop scheduling policy such as "static,2" is more effective, as it distributes both large and small chunks of work equally over all cores. Note that both of these programs are regular in the sense that their work profile does not depend on input data, thus static policies are most effective.



(a) Small problem size (N=160)



(b) Large problem size (N=1600)

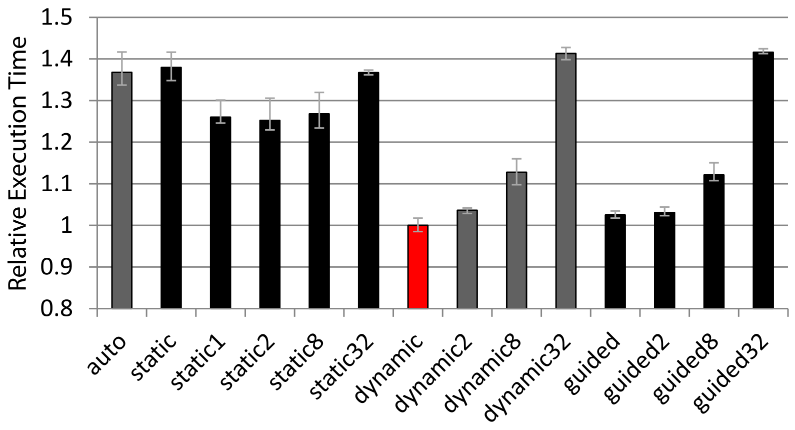
Figure 5.2: Initial experiments, impact of problem size.

5.2.2 Impact of Problem Size

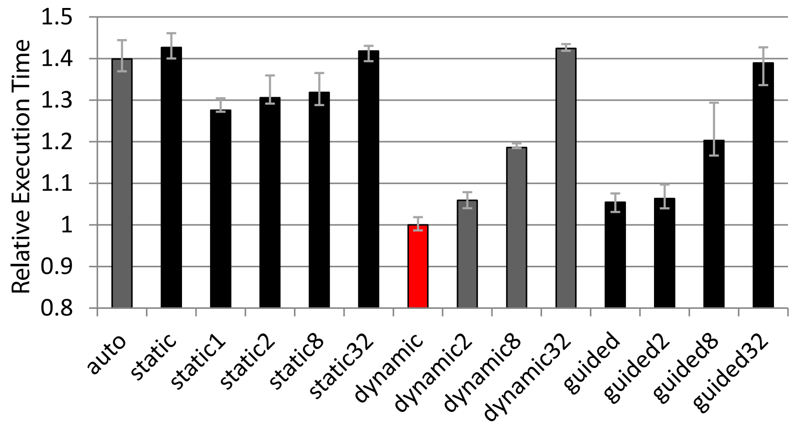
In the next experiment we investigated the impact of variations in the size of the problem / data set a program operates on on the ideal loop schedule. In Figure 5.2, the performance of the triangular matrix multiplication sample with two different problem sizes is compared. We see that with a small problem size, the negative performance impact of scheduling policies with a runtime component (dynamic, guided) increases, most likely due to thread scheduling overhead. Also, the increase in workload per chunk mitigates the slightly worsened load balance for a static chunk size of 8, leading to this configuration showing the best result.

With a larger problem size, the relative overhead of runtime scheduling is much smaller, tough still measurable. The round-robin static scheduling policy "static,1" features acceptable load balance with relatively low overhead, making it the best performing configuration.

5.2.3 Impact of External Load



(a) Low load (desktop) scenario



(b) High load (workstation) scenario

Figure 5.3: Initial experiments, impact of external load.

Finally, we look at a scenario that has often been neglected in loop scheduling research: the impact of external system load on the execution of a program. While this is an unusual situation in traditional HPC, where a cluster of servers is reserved for exclusive use by one program, it is the default on desktops, workstations and some large shared memory servers. With on-chip parallelism steadily increasing – even on embedded systems – and OpenMP being employed in end-user applications and games [49], we believe that an automatic loop scheduler needs to take this scenario into account.

Figure 5.3 shows the same program configurations as Figure 5.1(b) in two distinct load scenarios (for information on how the load simulation is performed, see Section 5.5). With increasing system load, more fine-grained runtime scheduling policies gain a significant advantage of up to 46%, compared to the default policy. These figures contain error bars since there was a slightly larger variance in the measurements – particularly for static scheduling – as a result of operating system scheduling behaviour.

To summarize, these initial findings guided the design of our loop scheduling in the following ways:

- As per the first set of figures, the automatic loop scheduler clearly needs to be aware of the *program structure*. This is accomplished via compiler analysis.
- However, as the second set of examples shows, just having static information is insufficient. The *problem size* is usually only known at runtime, which requires the integration of static compiler analysis with a runtime system.
- Finally, when exclusive use cannot be assumed, being aware of *external system load* is of utmost importance when selecting a scheduling policy. Thus, the runtime needs to consider the system state.

5.3 Related Work

Enhancing OpenMP loop scheduling is a topic that has been repeatedly investigated over the years. However, most research has focused on pure runtime solutions to the problem [86][6][91]. Conversely, our approach integrates an intelligent scheduling algorithm performed during runtime with meta-information provided by compiler analysis. Additionally, our runtime system takes into account external load, which is usually not considered in loop scheduling.

Recent work on compiler-based OpenMP loop scheduling by Wang et al. [88] uses machine learning to estimate the best loop scheduling policy at compile time. Since this is a pure compiler approach, it cannot deal with changing runtime conditions. Also, unlike the single-pass, symbolic analysis of our approach, it requires an extensive training phase.

The polyhedral model has been used in a few recent work related to OpenMP: some systems use OpenMP in conjunction with the polyhedral model to generate parallel code [17][10]. Others investigate its use in tool support by using information provided by polyhedral analysis of OpenMP programs to improve programmer error detection [12]. None of these works aim on improving loop scheduling by forwarding static analysis results to a runtime system.

5.4 Method

Our loop scheduling system consists of the following components:

- An advanced analysis component in the Insieme source-to-source compiler which generates a symbolic *effort estimation function* for each parallel loop in the target program, or a less accurate per-iteration effort value as a fallback.

- An extension to the Insieme-RS backend of the compiler which allows forwarding of this meta-information from the compiler to the Insieme runtime system.

- A monitoring component within Insieme-RS that measures the current external system load.

- An extension to the loop scheduling component of the Insieme runtime library, implementing a loop scheduling algorithm based on the meta-information provided by the compiler, the exact iteration range of the current loop and the external load.

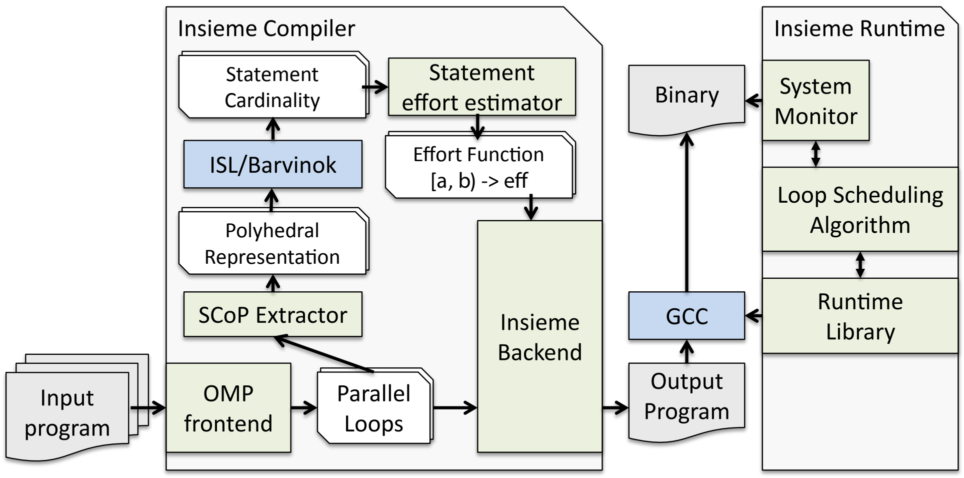


Figure 5.4: An Overview of the Architecture of our System

Figure 5.4 illustrates how these components interact on a high level. In the following subsections each component will be discussed in detail.

5.4.1 Compiler Analysis

The main goal of our compiler analysis is to obtain, for each parallel loop, an *effort estimation function* $f_{\text{effort}} \in \mathbb{N}^2 \to \mathbb{N}$. Given lower and upper iteration bounds a and b, the evaluation of $f_{\text{effort}}(a, b)$ provides an estimate for the computational cost of the corresponding subrange of the covered loop.

This effort estimation function is derived in several steps, starting from the parallel loop body B:

1. Enclose B in a *for* loop iterating over the symbolic range $[a, b)$.
2. Extract a polyhedral representation of this parameterized loop.
3. Set the effort estimation function $f_{\text{effort}}(a, b) := 0$
4. For each statement $stmt \in B$:
   (a) Use the barvinok [87] library to obtain a piecewise affine function for the statement's cardinality $f_{\text{card}}(a, b)$
   (b) Weight this function with the effort estimation $\text{eff}(stmt)$ for the statement, computing $f_{\text{stmt}}(a, b) := f_{\text{card}}(a, b) * \text{eff}(stmt)$
   (c) Add the statement effort to the total effort function $f_{\text{effort}}(a, b) := f_{\text{effort}}(a, b) + f_{\text{stmt}}(a, b)$
5. Algebraically simplify $f_{\text{effort}}(a, b)$ using CUDD [73]

In step 2, the internal representation (in INSPIRE) of the loop B is analyzed and a polyhedral representation is extracted. In-depth discussion of the polyhedral model and its application in compilers goes beyond the scope of this thesis – a thorough introduction is provided by Bastoul [11]. For our purpose, it suffices to mention that the polyhedral model can be applied to Static Control Parts (SCoPs). SCoPs are program regions that fulfill the following conditions:

1. All control structures are **for** loops or **if** statements with **affine** boundaries and conditions.
2. Arrays are the only complex data structures, and are accessed with affine subscript expressions.
3. Subscripts, bounds and condition expressions depend only on loop iterators and symbolic constants.

The polyhedral model assigns to each statement an n-dimensional polytope describing exactly how often it is executed within the modeled loop nest. Using this representation, a piecewise affine function expressing the number of executions of each statement can be calculated by computing its cardinality (4a). Note that the creation of a polyhedral representation of INSPIRE code was implemented by Simone Pelligrini, and that Herbert Jordan implemented the handling of symbolic algebraic functions in the Insieme compiler. Both of them are co-authors of the paper [82] in which this loop scheduling approach was initially published.

In step 4b we arrive at an effort estimation function for each such statement by weighting its cardinality function with an estimate for the cost of executing it once. The weighting factor eff(*stmt*) takes into account the expected number of CPU instructions and memory accesses required for the given statement. This estimation is rather simplistic in our current implementation: we count the number of memory accesses and floating point operations required to perform the statement in our internal representation, without taking into account any transformations performed by the back-end compiler. However, as only the *relative* effort of statements throughout the iteration space of each loop (that is, how much effort it is to execute a given sub-range of the iteration space compared to another sub-range of the same loop) is relevant for our loop scheduling system, this simple estimation is sufficient.

Special considerations apply when performing the SCoP analysis for our use case. Generally, the polyhedral model is used to *transform* code fragments (see Section 5.3), while we only use it to *estimate* effort. In the former case, the analysis needs to accurately cover all effects of the code to maintain the program semantics. For estimation, failing to fully analyze some statement means that the estimation function might be less accurate, potentially weakening the performance of the scheduling algorithm, but the program semantics are preserved. In practice, this allows us to extend the applicable range of our analysis by ignoring the side effects of external function calls, as long as we can provide an effort estimate for them (e.g. `printf`). We further extended the interprocedural applicability of our estimation by applying implicit inlining which is only done to calculate performance estimates, without affecting the generated code.

In the case where a loop can still not be covered by the polyhedral model despite these extensions, as is the case when control flow depends on input data, we make simplifying assumptions for loop boundaries and conditionals to generate a single scalar effort estimation representing one iteration of the parallel loop. Loops are assumed to have a 100 iterations, and conditions are assumed to be taken exactly half of the time they are encountered. Section 5.5.2 provides some experimental data on how commonly this fallback needs to be employed in real programs. Note that even this form of analysis failure provides useful information for the loop scheduler: if no static model can be established, it is more likely that the affected loop executes an input-dependent workload, and should thus be scheduled dynamically.

5.4.2 Compiler Backend

The Insieme compiler produces C code, which is in turn translated into a binary by a secondary compiler – typically GCC. As detailed in Section 3.4.3, the Insieme-RS compiler backend enumerates all the parallel loops included in the program, and, for each of them, generates a *work item structure*. To pass loop-related meta-information from the compiler to the runtime system, this structure is modified to include a (optional) function pointer of type `uint64 effort_estimator(int64 lower, int64 upper)` and a scalar fallback value `uint64 iteration_effort`. For each loop where our analysis was successful, the function pointer is set to a compiler generated C implementation of the deduced effort estimation function, otherwise it is set to `NULL` and the loop scheduling algorithm will use the fallback value.

5.4.3 Runtime Monitoring

The resource monitoring component of the runtime needs to measure the current *external load*, that is, CPU load generated by processes other than the managed parallel program. This is obtained by using the Linux `proc` filesystem. Specifically, the current processes' CPU usage values from `/proc/self/stat` are compared with the system-wide values obtained from `/proc/stat`, and a value between 0.0 and 1.0 representing the total external load across all cores is computed.

To minimize the overhead of this method and to increase measurement reliability, this value is cached and updated at most ten times per second. Increasing the update frequency did not improve scheduling performance in our experiments, and the overhead for performing the measurement at most ten times per second was proven to be negligible in our experiments (with a performance impact of less than 0.5%).

5.4.4 Loop Scheduling Algorithm

All information gathered by the components outlined above is used by the Insieme-RS loop scheduler to make a scheduling decision for each individual execution of every parallel loop. The decision procedure is outlined in Algorithm 5.1 and consists of four major steps:

1. Immediately schedule tiny loops if the estimated effort is small (lines 1-8)

2. Check the external load and use an adaptive dynamic schedule if it is greater than a threshold value (9-12)

3. If an effort estimator is available, use the calculated perfectly balanced distribution (13-15)

Algorithm 5.1 Automatic loop scheduling algorithm.

lower, upper	lower and upper bound of iteration range
members	number of members in the current communication group
estimator	effort estimation function for current loop
iter_effort	scalar per-iteration effort estimate for current loop
load	current external system load
MINEFF	minimum effort for consideration (constant per-system)
MINLOAD	minimum load for consideration (constant per-system)

```
1:  if estimator available then
2:      estimate = estimator(lower, upper)
3:  else
4:      estimate = (upper − lower) ∗ iter_effort
5:  end if
6:  if estimate < MINEFF then
7:      return immediate
8:  end if
9:  if load > MINLOAD then
10:     chunk = max((MINEFF/iter_effort) ∗ (1 − load), 1)
11:     return dynamic(chunk)
12: end if
13: if estimator available then
14:     shares = compute_shares(lower, upper, members, estimator)
15:     return balanced(shares)
16: else
17:     chunk = max(MINEFF/iter_effort, 1)
18:     return dynamic(chunk)
19: end if
```

4. Otherwise, assume irregular load and schedule dynamically (16-19)

The result of the algorithm determines the loop scheduling behaviour for the current loop execution instance. Three modes are available (see Section 3.3.3 for more information on and formal definitions of these modes):

immediate no parameters. Immediately executes the whole loop on the first work item to encounter it.

dynamic one parameter, the chunk size. Works like the standard OpenMP policy with the same name, dynamically distributing chunks of the loop range to requesting threads.

balanced requires an array of floating point values determining the relative size (in a fraction of the total number of iterations) of the shares for each member of the communication group. For example, [0.25, 0.25, 0.25, 0.25] would implement an equal distribution amongst four participants, while [0.6, 0.3, 0.06, 0.04] assigns progressively smaller chunks to subsequent members in the group.

The algorithm makes use of the compute_shares(`lower`, `upper`, `members`, `estimator`) function. It generates a balanced distribution that tries to assign approximately the same amount of work to each member of the current communication group. It first estimates the total effort for the given range [`lower`, `upper`], divides it by the number of work group members, and then uses a binary search to find a suitable chunk for each thread using the estimation function. Though this is usually a very quick process since the estimation function only takes a few cycles to run, the result is cached and reused if the same loop is executed for the same range again (memoization). This is a very common occurrence in HPC codes, and using memoization minimizes overhead in this case.

The parameters `MINEFF` and `MINLOAD` need to be set once per system. We have not yet developed a rigorous method for deducing these automatically. Nevertheless, experience indicates that systems are relatively insensitive regarding the precise values of these parameters, making them easy to tune manually.

5.5 Evaluation

In this section our system and algorithm are evaluated, starting with small kernels designed to allow easy analysis of the behaviour of the algorithm, followed by tests in a real-world setting. All experiments were performed on a SuperMicro 7046GT-TRF server with two Intel Xeon 5650 processors, containing 6 cores (12 hardware threads) each. The system runs CentOS version 5 (kernel 2.6.18) 64 bits. To compile the reference version of the example programs and as a secondary compiler for the code produced by Insieme, GCC version 4.5.3 was used with the -O3 flag set to reflect a production environment. When we refer to a "default" scheduling policy, we specifically mean the default implementation of the version of GOMP [60] included with this version of GCC.

To ensure statistical significance each experiment was repeated five times, and the median result is reported. In cases where significant statistical variance occurred vertical error bars are used to show the standard deviation. We depict three values per configuration (combination of program and system load state): the default OpenMP behaviour, the best result obtained using OpenMP policies for each configuration, and the result obtained by our method. The "best" OpenMP policy is found by exhaustive search across the following settings: [(no change), auto, static, dynamic, guided]. The latter three are tested with the chunk sizes 1, 2, 8 and 32. All values are normalized to the execution time of the best performing version.

External load profiles were recorded by monitoring each individual core of a reference system. During experiments, these profiles were replayed by a custom load generator. We used two separate load profiles, a "desktop" profile and a "workstation" profile. The former features generally lower load and short peaks of activity, while the latter shows a higher average load level and fully saturates some cores.

5.5.1 Kernel Experiments

For illustrative purposes, we will apply our method to three small kernels: a dense matrix multiplication, a triangular matrix multiplication, and a pendulum simulation. These represent three major classes of problems. Both the dense and triangular matrix multiplication satisfy the SCoP constraints and can therefore be rigorously analyzed. The former has a flat work profile and is thus ideally suited to static OpenMP scheduling, while the latter has a slanted work profile. Finally, the per-iteration work in the pendulum kernel strongly depends on the input data, hence it can not be covered by SCoP analysis.

Dense Matrix Multiplication

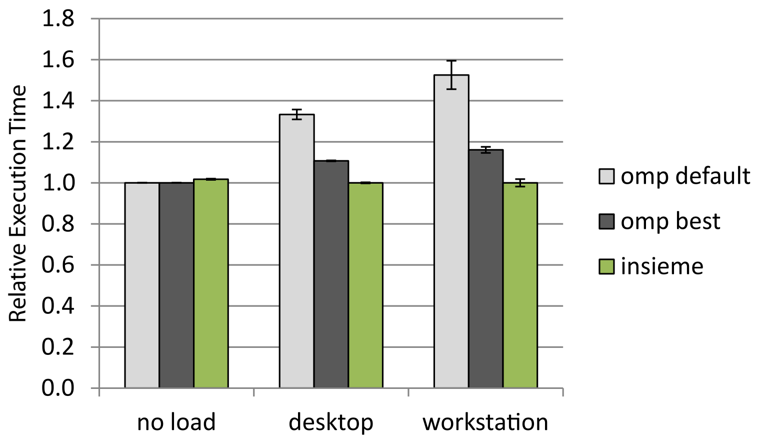


Figure 5.5: Dense matrix multiplication results.

Figure 5.5 shows the results for dense matrix multiplication. In the absence of external load, fully static scheduling is ideal for this kernel, and our implementation is 1.7% slower than the best (and default) OpenMP policy. With external load, the default policy is ineffective, and our result improves on the best OpenMP policy by 10% to 15%. The best policy found for desktop load is "dynamic,8" while the best policy for the workstation load profile is "dynamic". The reason for the good result demonstrated by our method is that due to the detection of external load the chunk size is adapted to fit the load profile at every point during the program's execution.

Triangular Matrix Multiplication

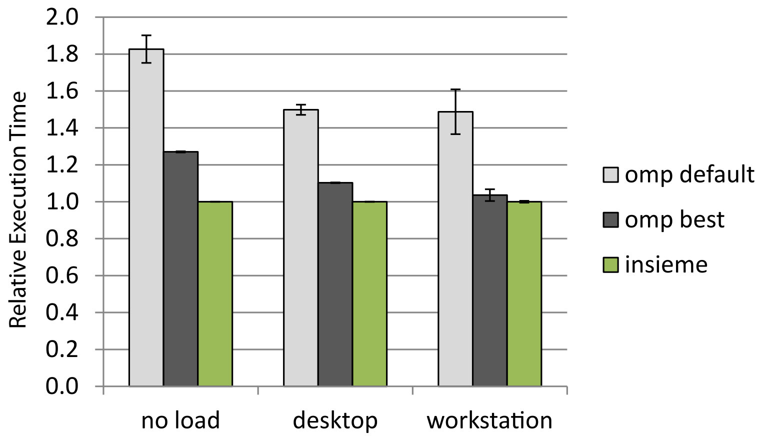


Figure 5.6: Triangular matrix multiplication results.

Next, we look at the triangluar matrix multiplication kernel, which has a more interesting load profile. As Figure 5.6 illustrates, the compiler-assisted workload distribution performed by our method in the unloaded case is very effective, improving performance by 82% compared to the default behaviour, and by 27% compared to the best OpenMP scheduling policy, “static,2”.

This improvement over the block-cyclic scheduling can be explained by the fact that even round-robin loop scheduling with a chunk size of 2 as performed by the “static,2” policy is not perfectly balanced. Furthermore, our scheduling distributes a single chunk to each thread, which induced less overhead and allows for better cache reuse than distributing many small chunks.

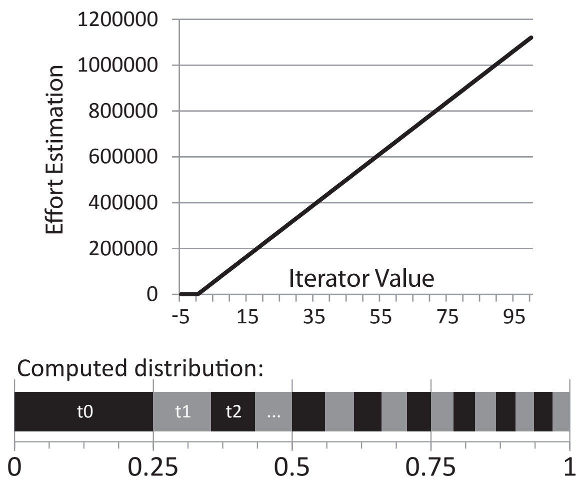


Figure 5.7: Triangular matrix multiplication effort estimation function.

The effort estimation function generated by our analysis for the triangular matrix multiplication test case and the per-thread shares computed for 16 threads in this scenario are shown in Figure 5.7. In the upper part, the effort estimation for each iterator value is plotted: iterations below zero perform no work, above that the amount of effort increases with the iterator value as the lower left triangular matrix rows become progressively wider.

For this test case, the best scheduling policy with a loaded system is “dynamic” for both load profiles. Our scheduling is the fastest for both situations, though in the “workstation” case the difference compared to the best OpenMP policy is negligible (3%).

Pendulum Simulation

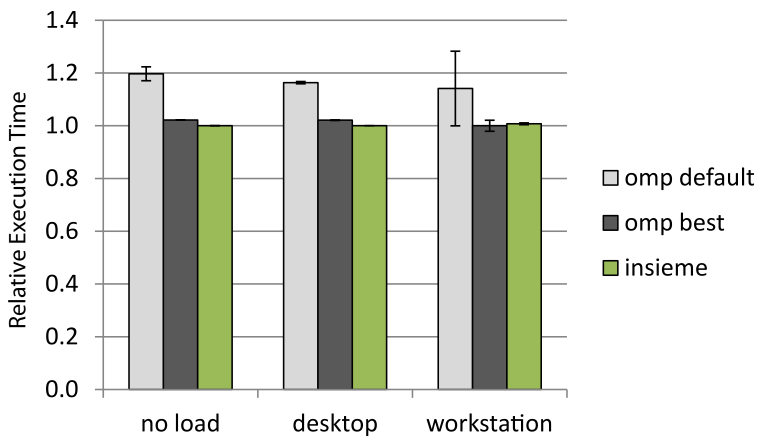


Figure 5.8: Pendulum simulation results.

The performance results for the pendulum kernel are depicted in Figure 5.8. This benchmark computes the resting points of pendulae under the effect of magnetic fields, from many starting locations. It is communication-free but has an unpredictable, input data dependent load imbalance, causing default scheduling to be sub-optimal. For the case with no load, the “dynamic,2” policy is best, while for the other two cases “dynamic” performs best.

When the workstation external load profile is active, our method performs slightly (0.7%) worse than the “dynamic” OpenMP policy. For this load profile and the loop effort estimated for this kernel, our scheduler always decides to dynamically distribute a single loop iteration, thus performing exactly the same operation as the “dynamic” policy. The 0.7% difference can be explained by the overhead introduced by our scheduling process.

5.5.2 Real-world Applicability

While the results measured on small kernels are encouraging, methods based on extensive compiler analysis often fail when applied to larger code bases. However, the polyhedral model has been successfully used in production compilers [84], and, as described in Section 5.4.1, we were able to further relax some of its constraints for our use case.

In this section, we present an experimental analysis on some of the benchmarks contained in the NAS Parallel Benchmarks (NPB) [9] suite. As a first step we investigate the extent to which the parallel loops contained within these programs can be treated with our analysis method.

Table 5.1: Applicability of our analysis on NPB loops.

State	Number of loops	% of loops
Total	465	100.0%
Fully analysed	373	80.2%
Non-affine expressions	57	12.3%
Data-dependent control flow	33	7.1%
Contain while loops	2	0.4%

Table 5.1 lists total number of loops contained within the NPB programs, the amount that were fully analysed, and groups those that could not be analysed into categories depending on the reason for the analysis failure. Note that the number of loops listed here is higher than the amount statically contained within the program source code, due to our method analysing each call site separately.

More than 4 out of 5 of all parallel loops contained in the set of benchmarks can be analyzed. The most common reason for analysis failure are non-affine boundary, condition or subscript expressions, followed by data-dependent control flow. Two of the parallel loop nests contain *while* loops.

The results of our performance evaluation are summarized in Table 5.2. The "Default" and "Best" columns list the relative difference in execution time achieved by our scheduling system compared to default scheduling (as specified by the benchmarks) and the best scheduling policy found in the search space described earlier. For example, 4.2% in the ft.B/none/default cell means that executing the ft benchmark with no external load and the default scheduling policy took 104.2% of the time the same configuration took using our scheduling system.

Predefined problem size B was chosen for all the benchmarks as a good compromise between realistic size and maintaining a feasible duration for the experiments. The **GM** values are the geometric means, for each configuration, across all benchmarks, and Figure 5.9 illustrates these values.

Table 5.2: Nas Parallel Benchmark performance results.

Name	External Load	Gain Over Default	Gain Over Best	Best Config
ft.B	none	4.2%	-0.2%	static,1
ft.B	desktop	21.8%	4.4%	dynamic,2
ft.B	workstation	59.9%	11.2%	dynamic
ep.B	none	14.0%	-1.9%	dynamic,8
ep.B	desktop	3.2%	-0.9%	dynamic
ep.B	workstation	19.7%	3.0%	dynamic,32
bt.B	none	-2.4%	-2.4%	static
bt.B	desktop	70.8%	65.2%	dynamic
bt.B	workstation	*	*	*
cg.B	none	8.4%	3.9%	guided,32
cg.B	desktop	113.4%	111.2%	guided,32
cg.B	workstation	471.3%	451.7%	guided,8
mg.B	none	51.7%	5.3%	dynamic
mg.B	desktop	56.1%	33.0%	dynamic
mg.B	workstation	157.4%	110.8%	dynamic,2
GM	none	13.7%	0.9%	
GM	desktop	48.2%	36.8%	
GM	workstation	94.9%	67.7%	

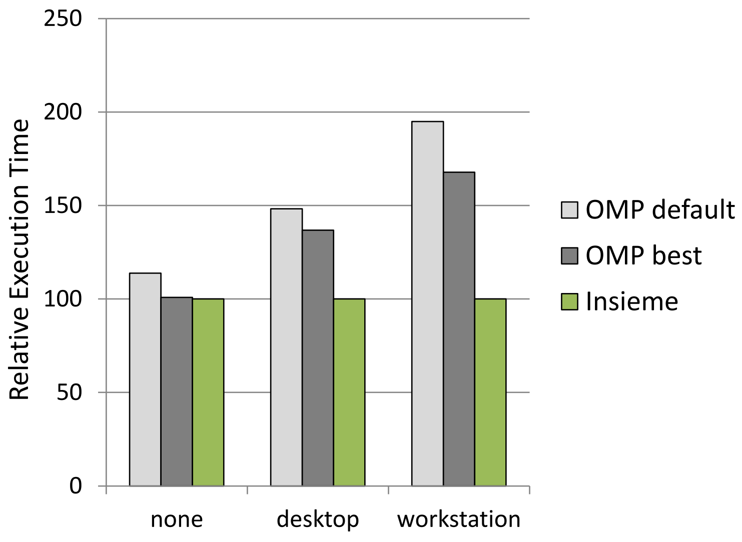


Figure 5.9: Geometric mean of real-world benchmark results.

Some points that deserve particular attention are:

- The bt benchmark with workstation external load could not be completed due to time constraints – the execution time increased disproportionately with increased load across all scheduling policies.
- There is only a single case where our algorithm performs worse than the default: bt with no load. It is the only benchmark where the default scheduling (static) is also the best policy. For most loops within bt our method picks this optimum, but for one of them the analysis fails, causing a fallback to a slightly less efficient dynamic schedule.
- The best speedup in a load-free scenario occurs for mg. This is due to the nature of the algorithm implemented by this benchmark, which leads to some loops being executed with very small iteration domains. These are identified as low-effort by our method and immediately scheduled as a whole on the first thread available.
- Generally, higher levels of external load favour our system, which can effectively adapt to them.
- Even with no external load, our method tends to achieve a marked improvement over default scheduling due to the availability of compiler-deduced meta-information. The average speedup obtained in this setting is 13%.

5.6 Summary

This chapter presents an automatic OpenMP loop scheduling method which combines advanced compiler analysis with a load-aware runtime system, leveraging the strengths of the Insieme compiler infrastructure and its integration with Insieme-RS. Polyhedral analysis is used to calculate a parameterized *effort estimation function* for each parallel loop, based on the cardinality of all statements it contains. Executable code for this function is generated by the compiler backend, and invoked at runtime to calculate an ideal balanced schedule or estimate efficient chunk sizes for dynamic scheduling. Additionally, external CPU load is taken into account during the scheduling process.

We evaluated our system on small kernels as well as programs from the NAS Parallel Benchmarks suite, and achieved improvements of up to 82% in the unloaded state, and 471% with heavy external load, compared to default OpenMP scheduling.

To estimate the absolute effectiveness of our approach, we performed an exhaustive search over a broad range of standard OpenMP scheduling policies and compared with the best results. Our scheduling frequently improves upon even this tuned result, particularly in scenarios featuring external load. The results of our automatic scheduling algorithm are stable across a wide range of programs and execution environments: in fact, the worst-case performance achieved by our fully automatic approach is within 3% of the best standard OpenMP policy.

Chapter 6

Optimizing Granularity in Task-based Parallelism

6.1 Introduction

Task-based parallelism is one of the most fundamental parallel abstractions in common use today [4]. While relatively easy to implement and use, achieving good efficiency and scalability with task parallelism can be challenging. A central feature of every task-based parallel program that significantly affects both efficiency and scalability is *task granularity* [32]. The *granularity* of tasks is defined by the length of the execution time of a single task between interactions with the runtime system, such as spawning new tasks.

Very fine-grained, short-running tasks lead to a loss in efficiency compared to sequential execution due to the runtime overhead associated with generating and launching a task, as well as synchronizing its completion with other tasks in the system. On the other hand, coarse-grained, long-running tasks minimize overhead, but are hard to schedule effectively and may therefore fail to scale well on large parallel systems. Previous work in this area has focused mostly on runtime systems or user-controlled cutoffs to manage granularity (see Section 6.3). Conversely, we propose an approach that combines a multiversioning compiler with a runtime system which adaptively selects from the generated versions. Our goal is to maximize efficiency by increasing task granularity – and thus decreasing overheads – without negatively affecting load balance or scalability.

We implemented our method for OpenMP [65] tasks within the Insieme compiler and runtime system, but the idea is equally applicable to any other task parallel language. Our concrete contributions are the following:

- A compile-time multiversioning transformation that generates a set of task implementations of increasing granularity by recursive *task unrolling* and subsequent elimination of superfluous synchronization primitives. This transformation is applicable to both simple recursion and N-ary mutual recursion.

- A runtime heuristic for the dynamic adaptation of granularity based on the concept of *task demand*, which automatically choses the code version to execute at each task spawning point.

- Evaluation and analysis of the performance of our method on a number of well-known task parallel benchmarks. We compare with other OpenMP implementations, our own implementation without the multiversioning optimization and Cilk [15] versions which represent the state of the art in fine-grained task parallelism.

In this chapter, we use the word *task* to generically refer to any way of implementing tasks, e.g. when describing operations or overheads that are part of any such implementation. In Insieme-RS, tasks map to work item instances, as laid out in Section 2.3.2. Therefore, we refer to *work items* when we describe the specific implementation within our system.

The remainder of this chapter is structured as follows. In Section 6.2 we provide some initial results which motivated our approach, followed by an overview of related work in Section 6.3. We then describe our method in detail in Section 6.4 and evaluate its performance in Section 6.6. Finally, Section 6.7 summarizes and concludes our findings.

6.2 Motivation

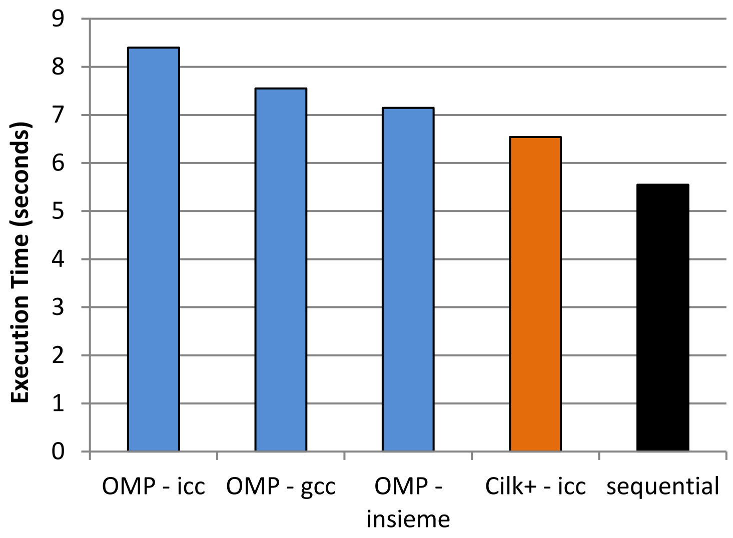


Figure 6.1: Initial Task Experiments, N-Queens $N = 13$, Single-threaded.

In this section we present some initial benchmark results that motivate our multiversioning method. Figure 6.1 shows single-threaded execution times measured for the Barcelona OpenMP Tasks Suite (BOTS) [31] N-Queens benchmark with $N = 13$. For details on the hardware, compiler versions and programs used refer to Section 6.6.

The lowest execution time amongst the OpenMP versions is achieved by our compiler and runtime system (Insieme), however, this time is still 28% higher than purely sequential execution. Even the Cilk version, while more efficient than any OpenMP implementation, is 19% slower than the sequential version. Our multiversioning method is designed to address this inefficiency. Throughout this chapter, when we refer to *inefficient* execution, we mean execution which takes longer than executing purely sequential code (assuming ideal speedup as defined in Section 2.2.2).

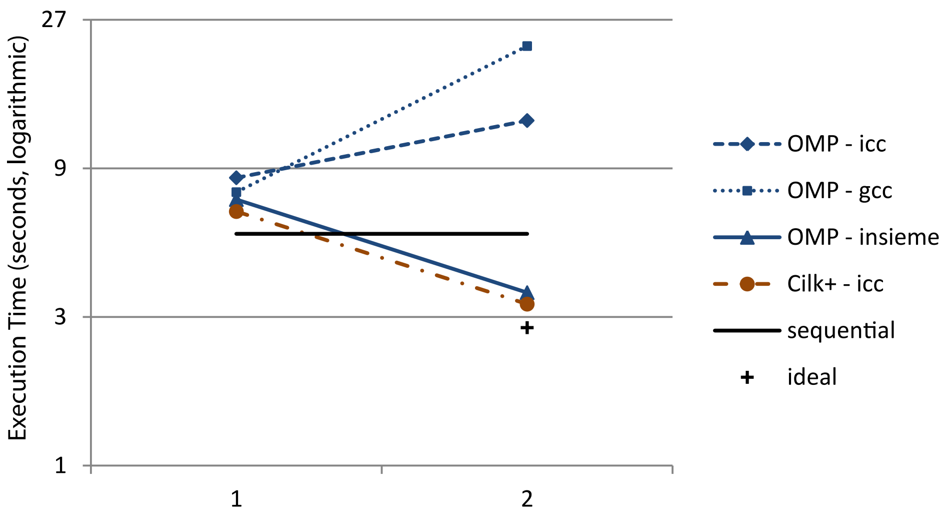


Figure 6.2: Initial Task Experiments, N-Queens $N = 13$, Scaling from 1 to 2 threads.

Note that the OpenMP runtime systems of ICC [40] and GCC [74] perform special case handling when only a single worker thread is used. This is visible in Figure 6.2, which shows their performance degrading when switching from one to two threads. Further experiments in Section 6.6 confirm this behavior, with scaling starting after some initial performance degradation when activating multi-threaded execution. The OpenMP version compiled with Insieme and the Cilk version do not suffer from this issue, however they still induce a relative overhead of about 20% compared to ideal linear scaling from the sequential version. We identified the following potential causes for this inefficiency:

1. Task generation overhead. This includes generating a task structure,

populating it with values and enqueuing it. In Insieme-RS, it equals the creation of a work item instance, as defined in Section 2.3.2, and the allocation of its required data items.

2. Synchronization primitive overhead (e.g. `taskwait`). At the very least, this involves keeping track of all the subtasks launched by each task, and signaling when they are complete. (Performed by the event handling system in Insieme-RS, see Sections 3.3.2 for the definition of its semantics and Section 3.5.3 to understand the associated overhead costs)

3. Task library calls. The runtime methods required for tasking are generally implemented in a separate library, and the overhead for their invocation is incurred even if they perform no actual work.

4. Non-inlineable, indirect program function calls. Since the program function implementing a given task needs to be called by the tasking library, a pointer to it is usually passed to the library function. Even if the runtime library decides to directly execute the call, this prevents the benefits – improved instruction scheduling and a reduction in overhead – associated with inlining.

Issues 1 and 2 can be mitigated by a pure runtime approach, e.g. the runtime library can dynamically decide whether to generate a full task structure or directly call the task function. This method is usually referred to as lazy task creation [57]. However, the basic overhead of library function calls (issue 3) and the fact that indirectly called functions in the original program can not be inlined (issue 4) can not be changed at runtime and need to be handled at compile time. This limitation of pure runtime systems motivates our compiler-aided multiversioning approach.

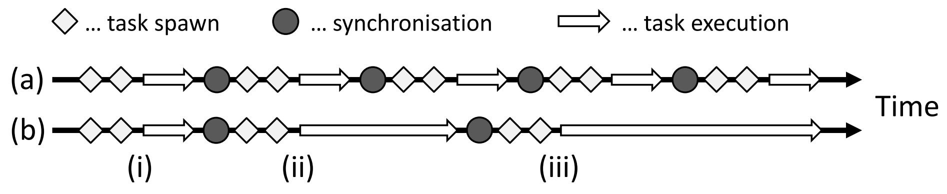


Figure 6.3: Timelines for default task execution and execution with variable granularity.

All four potential causes for inefficient execution identified above are directly related to and influenced by the granularity of tasks. The more often individual tasks are generated and synchronized, the higher the impact of the associated overheads on execution time. However, simply increasing the granularity of all tasks is not a solution: such an approach will lead to load imbalance, and it increases the probability of workers idling. Therefore, our goal, as illustrated in Figure 6.3, is the generation of different implementations for each task. The upper timeline (*a*) shows the default execution of a single Insieme-RS worker in a task-parallel program. All tasks have the same granularity and execution time. The lower timeline (*b*) depicts our ideal goal, involving dynamic selection from a set of implementations at each task spawning point. Early on, at point (*i*), a fine-grained task is generated so that the desired degree of parallelism is achieved quickly. At later points (*ii*) and (*iii*) the system is saturated and therefore the task granularity is gradually increased, reducing the inherent overhead caused by any interactions with the runtime library.

6.3 Related Work

Much previous work on parallel tasks has focused on runtime systems [19] or scheduling policies [64]. As described in section 6.2, pure runtime modifications are incapable of dealing with all the causes for inefficiency that our combined compiler and runtime approach covers. Moreover, our proposed multiversioning scheme is orthogonal to task scheduling decisions and can be combined with any scheduling policy.

A common approach towards dealing with task granularity issues is having the user provide thresholds or cut-off values [32]. In our work, task granularity is controlled entirely by the compiler and runtime system, without requiring manual programmer support. Duran et al. [29] describe an adaptive cut-off method which does not require manual adjustment, but their pure runtime approach does not offer the performance benefit of full sequentialization in the compiler.

Inlining of recursive functions has been previously performed in sequential program transformation [35], even with the express purpose of improving performance in divide and conquer programs by reducing overheads [69]. However, these works do not deal with parallelism, while our approach focuses primarily on minimizing the overhead incurred by parallel task creation and synchronization.

Some recent publications have used compiler multiversioning in a parallel setting [26][42], but they focused exclusively on loop-based data parallelism. Conversely, our multiversioning approach is designed for task-parallel, recursive programs.

Very recently, Deshpande and Edwards used recursion unrolling to improve opportunities for parallelism in Haskell programs [27]. Unlike our method, they do not use multiversioning or version selection at runtime, and their compiler transformations are designed for the Haskell functional language while we process input programs written in C with OpenMP.

6.4 Method

In this section our task multiversioning method is described. Section 6.4.1 provides an overview of our central idea, and how the compiler and run-time components work together in order to control task granularity. Section 6.4.2 details the compiler transformations used during the multiversioning process, while 6.4.3 describes the scheduling heuristic employed in the run-time system.

6.4.1 Overview

Figure 6.4 illustrates the major components of our proposed method. Starting from an OpenMP program with parallel tasks, our compiler generates an Insieme-RS client application in which multiple different implementation versions of each task are encoded. During execution of the program, whenever a specific task is invoked, Insieme-RS selects and launches a version of this task.

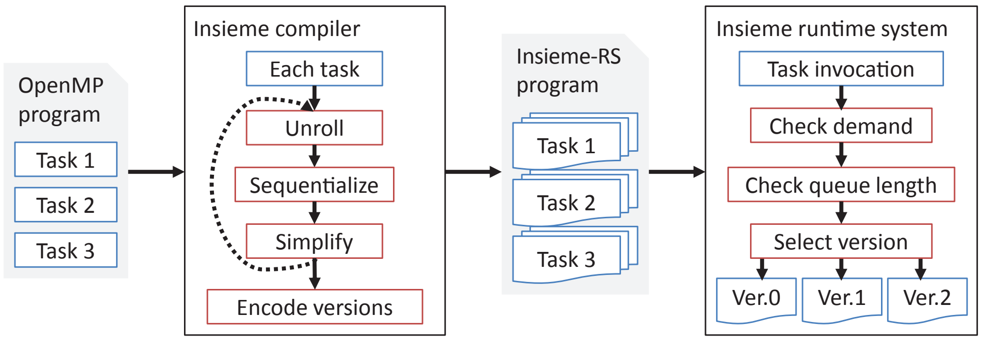


Figure 6.4: Overview of our Method

6.4.2 Compile-time Multiversioning

During compilation our goal is to generate multiple versions of each parallel task, with varying granularity. As depicted in Figure 6.4 this involves a three step process, which may be applied multiple times to further increase the task size. The individual steps are as follows:

1. **Task unrolling**. Replaces each task invocation site with a direct call to the task function, which is subsequently inlined. This can be thought of as a context and parallelism-aware recursive function inlining step. The name *task unrolling* is adapted from Rugina's usage of *recursion unrolling* [69].

2. **Sequentialization**. This step focuses on identifying which synchronization primitives – if any – were rendered superfluous by the partial

elimination of parallel task invocations due to task unrolling, and removing them. It is described in more detail below.

3. **Simplification**. The unrolling and sequentialization may have generated code that can be simplified by basic optimizing compiler transformations such as arithmetic simplification, constant propagation or dead code elimination. Thus, these are performed before any further processing.

The number of generated versions depends on the granularity of the initial tasks and the largest granularity desired. The versions are generated and encoded into the output program in the following order.

1. **Original**. The original version from the input program.

2. ***N* times unrolled versions**. Starting from $N = 1$. In these versions, only partial sequentialization is performed. Outer task spawning points are removed, but the innermost spawning location is kept. This process is illustrated in detail in a code example in Figure 6.6, described below.

3. **Fully sequentialized version**. In this version all task spawning points are removed and replaced with plain function calls.

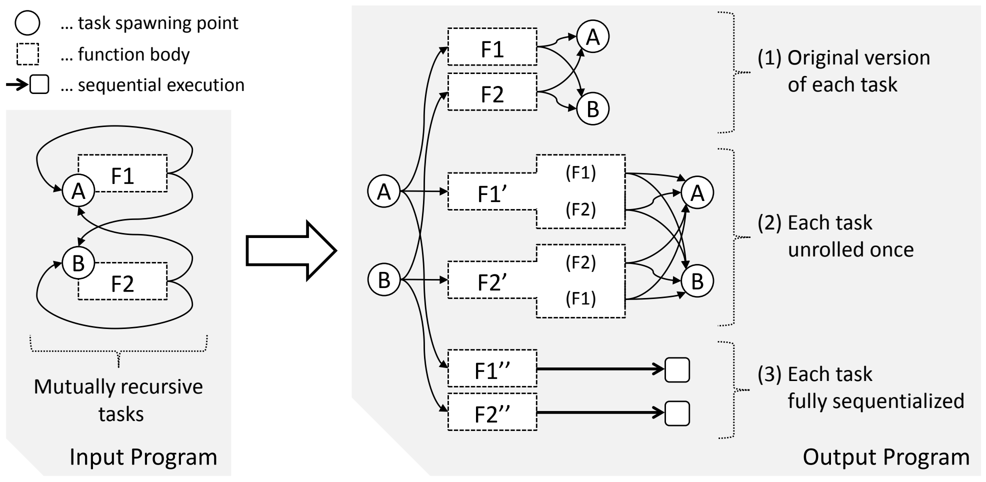


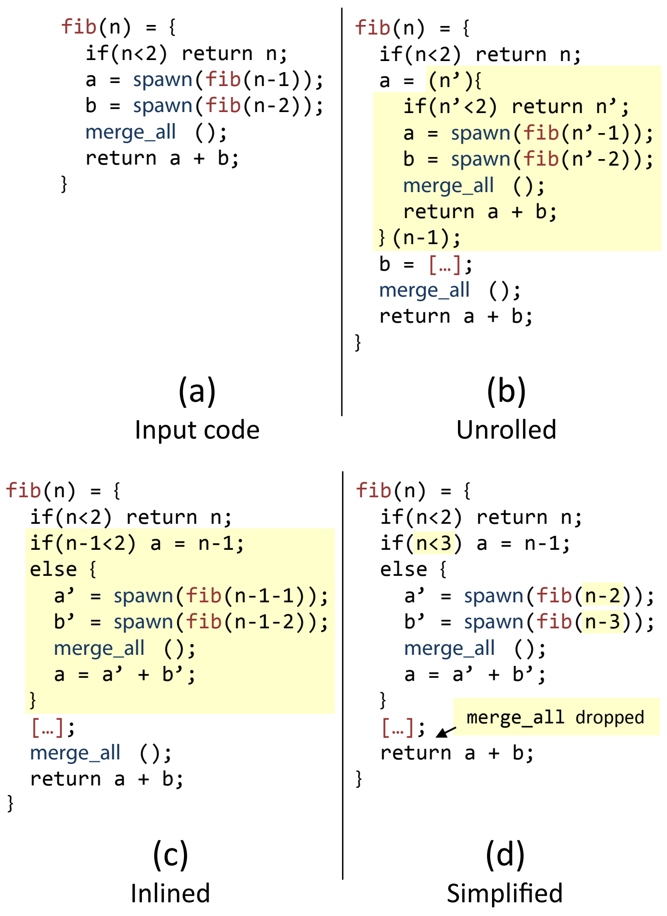
Figure 6.5: Version Generation and Control Flow.

Figure 6.5 illustrates the result of generating 3 versions for a mutually recursive task set consisting of two functions $F1$ and $F2$. The original program thus has two task spawning locations, A (which spawns $F1$) and B (spawning $F2$). In the parallel program model introduced in Section 2.2.2,

$\{A, B\} \subseteq \Psi$, that is, A and B are nodes which generate a new parallel control flow. To improve the clarity of the illustration, these task spawning points have been replicated in the figure, however they are still all referring to the same task.

Version (1) is identical to the original program, except that at each spawning point there is now a choice between 3 distinct implementations of each function. In version (2), consisting of $F1'$ and $F2'$, each recursive task invocation was unrolled once, forming tasks of increased granularity. Clearly, if this version is used, more work is performed between individual task invocations and interactions with the runtime library. Finally, version (3), comprising $F1''$ and $F2''$, is fully sequentialized. Once this version is invoked, no further parallel tasks will be spawned on this branch of the recursive descent.

Code Example



```
fib(n) = {
  if(n<2) return n;
  a = spawn(fib(n-1));
  b = spawn(fib(n-2));
  merge_all ();
  return a + b;
}
```

```
fib(n) = {
  if(n<2) return n;
  a = (n’){
    if(n’<2) return n’;
    a = spawn(fib(n’-1));
    b = spawn(fib(n’-2));
    merge_all ();
    return a + b;
  }(n-1);
  b = […];
  merge_all ();
  return a + b;
}
```

```
fib(n) = {
  if(n<2) return n;
  if(n-1<2) a = n-1;
  else {
    a’ = spawn(fib(n-1-1));
    b’ = spawn(fib(n-1-2));
    merge_all ();
    a = a’ + b’;
  }
  […];
  merge_all ();
  return a + b;
}
```

```
fib(n) = {
  if(n<2) return n;
  if(n<3) a = n-1;
  else {
    a’ = spawn(fib(n-2));
    b’ = spawn(fib(n-3));
    merge_all ();
    a = a’ + b’;
  }
  […];
  return a + b;
}
```

Figure 6.6: Example task transformation - Fibonacci - Version generation.

Figure 6.6 illustrates the effect of the steps taken during compilation to generate a task version that has been unrolled once. A pseudo-code formulation is used for reasons of clarity and size. It is C-like, but without the need for explicit type specification, and with two additional keywords: `spawn` implies the generation of a new parallel task (corresponding to `#pragma omp task untied` in OpenMP and a work item `spawn` operation in Insieme-RS), while `merge_all` waits for the completion of all launched subtasks (equivalent to `#pragma omp taskwait` and a work item `join_all` operation).

In (a), the original input code is shown. Moving on to (b), first-level task invocations are removed and replaced with in-place calls of the associated functions. Context-sensitive inlining of these calls results in (c). Finally, redundant applications of the `merge_all` operation are removed and arithmetic simplification is applied. The final generated code for this version is listed in (d). This process can be repeated N times to generate increasingly larger task sizes.

```
fib(n) = {
  if(n<2) return n;
  a = spawn( pick(
    fib(n-1),
    fib_u1(n-1),
    fib_seq(n-1) ) );
  b = spawn( pick(
    fib(n-2),
    fib_u1(n-2),
    fib_seq(n-2) ) );
  merge_all();
  return a + b;
}
```

(a)
Original

```
fib_u1(n) = {
  if(n<2) return n;
  if(n<3) a = n-1;
  else {
    a’ = spawn( pick(
      fib(n-2),
      fib_u1(n-2),
      fib_seq(n-2) ) );
    b’ = spawn(pick(…));
    merge_all();
    a = a’ + b’;
  }
  […];
  return a + b;
}
```

(b)
Unrolled Once

```
fib_seq(n) = {
  if(n<2) return n;
  if(n<3) a = n-1;
  else {
    a’ = fib_seq(n-2);
    b’ = fib_seq(n-3);
    a = a’ + b’;
  }
  […];
  return a + b;
}
```

(c)
Fully Sequentialized

Figure 6.7: Example task transformation - Fibonacci - Generated versions.

After all the versions are generated, each version needs to be modified to enable runtime selection. Figure 6.7 contains the final code for the original version with task selection (a), the unrolled version as discussed previously (b) and a fully sequentialized version (c).

The `pick` keyword implies a possible choice between semantically equivalent versions, which is deferred to the runtime system. Its semantics are equivalent to those of the θ node introduced in our model of parallel programs with nondeterministic choice (Definition 5). This choice is included at the spawning points of the original version, as well as all unrolled versions. In the fully sequentialized version, the spawning point is removed and replaced with a direct recursive call to the sequentialized function.

Partial Sequentialization

In most parallel programs there will be some superfluous synchronization statements after task unrolling. Since the execution has been partially sequentialized, instructions that wait for the completion of a task that was unrolled are no longer necessary and should be removed. The transformation eliminating unnecessary synchronization acts as detailed in Algorithm 6.1 on a task version T, effectively removing all `merge_all` operations for which there is no possibility of any task being spawned between them and a previous `merge_all`.

Algorithm 6.1 Superfluous Synchronization Elimination Algorithm.

```
    T                                  input/output task version

 1: Determine the set $M$ of all merge_all invocations in $T$.
 2: for all merge_all $m \in M$ do
 3:     Compute the set of all execution paths $F$
            from the entry point of $T$ to $m$.
 4:     for all paths $f \in F$ do
 5:         Reverse $f$ and remove the first entry.
 6:     end for
 7:     if $\forall f \in F : f$ encounters no spawn before a merge_all then
 8:         Remove $m$ from $T$.
 9:     end if
10: end for
```

Superfluous Synchronization Elimination Example

As an example, Algorithm 6.1 is applied to the task version generated in Figure 6.6 (c). The full code for this stage in the version generation is given in Listing 6.1, and we will refer to the code statements by their line number, as well as the labels added for spawn and merge operations. Computing the set M as per the algorithm for this example yields $M = \{m_a, m_b, m_1\}$.

For m_a, the set $F = \{(1, 5, 6, 7)\}$. Reversing the single included path and removing the first entry results in $F = \{(6, 5, 1)\}$. The statement at line 6 is s_{a2}, a `spawn` operation, thus m_a is kept. For m_b the situation is similar, and a `spawn` operation is encountered immediately on the reversed paths, but the paths are slightly more complex.

Finally, consider m_1. In this case, the initial set of paths is given by

$$\begin{aligned} F \ = \{ \quad & f_0 = (1, 5, 6, 7, 8, 12, 13, 14, 15, 17), \\ & f_1 = (1, 3, 12, 13, 14, 15, 17), \\ & f_2 = (1, 5, 6, 7, 8, 10, 17), \\ & f_3 = (1, 3, 10, 17) \qquad \} \end{aligned}$$

Reversing each path and removing the first entry results in

$$F = \{ \begin{aligned} & f_0 = (15, 14, 13, 12, 8, 7, 6, 5, 1), \\ & f_1 = (15, 14, 13, 12, 3, 1), \\ & f_2 = (10, 8, 7, 6, 5, 1), \\ & f_3 = (10, 3, 1) \end{aligned} \}$$

On the path f_0, the **`merge_all`** operation m_b is encountered at 14, before any **`spawn`**. On f_1, the situation is the same. On f_2, the **`merge_all`** operation m_a is encountered at 7, again before any **`spawn`**. Finally, no **`spawn`** operation is contained in f_3. Thus, the condition holds for all paths and m_1 can safely be eliminated.

```
1  fib(n) = {
2    if(n<2) return n;
3    if(n-1<2) a = n-1;
4    else {
5      a  = spawn(fib(n-1-1));        s_a1
6      b  = spawn(fib(n-1-2));        s_a2
7      merge_all();                   m_a
8      a = a + b;
9    }
10   if(n-2<2) b = n-2;
11   else {
12     a   = spawn(fib(n-2-1));       s_b1
13     b   = spawn(fib(n-2-2));       s_b2
14     merge_all();                   m_b
15     b = a  + b  ;
16   }
17   merge_all();                     m_1
18   return a + b;
19 }
```

Listing 6.1: Synchronization Elimination Code Sample.

6.4.3 Runtime Version Selection

The previous section outlined how multiple versions with different granularities and trade-offs are generated in the compiler. This provides the runtime system with an opportunity of making a version choice every time a task is spawned. Making the wrong choice can result in reduced efficiency, or, at worst, greatly diminish parallelism – e.g. in case a fully sequentialized version is chosen too early. We considered the following design goals and observations when developing our version selection method:

- At the start of the program, the original (most fine-grained) version of the tasks should be used, since the parallelism available in the system is not yet fully leveraged and load-balancing is a priority.

- The impact of conservative behavior – i.e. using more fine-grained tasks – causes more gradual performance degradation than using tasks that are too coarse grained, potentially leading to some worker threads idling.
- The decision procedure needs to be simple, causing only little overhead, otherwise it could negate any benefits from multiversioning.
- The decision making process should be distributed – no new synchronization points between worker threads should be introduced to facilitate version selection.

Taking these points into account led to the development of a distributed version selection heuristic based on two parameters that are tracked for each individual Insieme-RS worker. The first parameter is *task demand*, which keeps track of other worker's unfulfilled attempts to steal tasks from the local worker. The second parameter is the *queue length* of each worker, which indicates how many tasks it currently has available for execution or stealing.

Task demand is tracked in a surprisingly simple, but effective, manner. The demand is stored as an integer which starts at a positive value equal to the maximum task queue length. Whenever a task is generated by a worker thread, it reduces its own task demand value by 1. When a worker k_1 attempts to steal from another worker k_2 which has no tasks available, then the task demand value of k_2 is reset to the maximum task queue length.

Algorithm 6.2 Task Version Selection Algorithm.

`queue_length`	current queue length
`task_demand`	current task demand
`num_versions`	number of versions generated for current task
`MAX_QUEUE`	maximum queue length (fixed)

output:	0	$\Leftrightarrow$	original task
	$N = 1 \dots \texttt{num\_versions} - 2$	$\Leftrightarrow$	unrolled N times
	$\texttt{num\_versions} - 1$	$\Leftrightarrow$	fully sequentialized

```
1: version = num_versions − ⌈(task_demand/MAX_QUEUE) ∗ num_versions⌉
2: if version >= num_versions − 1 then
3:     if queue_length == MAX_QUEUE then
4:         return num_versions − 1
5:     end if
6:     return num_versions − 2
7: end if
8: return version
```

Our version selection procedure is listed in Algorithm 6.2. In conjunction with the demand tracking outlined above, it has the following desirable properties:

- Evaluating the selection function only takes a few dozen cycles, assuming that all the required values are cached.
- The way in which task demand is reset to the initial value if any work item stealing operation fails, but is only reduced gradually during normal execution, mirrors the earlier observation about the negative performance impact of wrong granularity selection. It makes the expensive case of idle workers unlikely by reacting very strongly to failed stealing attempts.
- Selecting the fully sequentialized version is a step that should only be taken after careful consideration, since it will prevent any further parallelism from being generated on this branch of the recursive descent. Therefore, the heuristic only takes this step if there has been no demand for additional tasks over a large number of spawn points *and* the queue is full.

The choice of the `MAX_QUEUE` parameter has an impact on the effectiveness of this approach. Experimental evaluation has shown that generally, a longer queue is beneficial on systems with a larger number of cores. For the evaluation in Section 6.6, `MAX_QUEUE` was set to 32.

Task Version Selection Example

Let us assume for this example that 4 code versions were generated for a given work item corresponding to a task, that is `num_versions` $= 4$. Given the mapping in Algorithm 6.2, this means that version 0 is the original code, in version 1 recursive task invocations have been unrolled once, in version 2 they have been unrolled twice and version 3 is fully sequentialized.

Now, assume two workers k_1 and k_2, and `MAX_QUEUE` $= 4$ (a very low value chosen for illustrative purposes). Both workers start with a `task_demand` $t^d_{k_1} = t^d_{k_2} = 4$. Let us now investigate the execution of the simple fibonacci code sample used previously in this setting, with k_1 starting the outermost task execution. At the start of the program, `version` $= 4 - \lceil(4/4) * 4\rceil = 0$, which means that the initial code version (with the smallest granularity) is chosen. This results in the spawning of two new work items, decrementing $t^d_{k_1}$ by 1 each, resulting in $t^d_{k_1} = 2$. At this point, k_2 may or may not steal work items from k_1 – it does not change the further execution unless the queue in k_1 becomes empty. If that happens, $t^d_{k_1}$ gets reset to 4. Let us assume for this example that this does not occur.

When k_1 selects a follow-up task version, the selection algorithm will evaluate to `version` $= 4 - \lceil(2/4) * 4\rceil = 2$. Thus, the 2 times unrolled version is selected, which generates 8 new work items. This will fill up the queue and set $t^d_{k_1} = 0$. At this point, as long as the queue remains full, `version` >= `num_versions` − 1 && `queue_length` == `MAX_QUEUE` will evaluate to true, and the next task will be executed with full sequentialization. As soon as k_1's queue loses an element, either because it is stolen by k_2 or launched by k_1 itself, it will fall back to executing the unrolled but not fully sequentialized code version 2, immediately refilling the queue. Thus, unless a stealing attempt fails later on, this particular program will complete using primarily the highly efficient fully sequentialized versions, with some interspersed partially unrolled versions.

6.5 Implementation

We will now describe how the method outlined above was implemented in our compiler and runtime system. The compiler transformations implemented for this work are detailed in Section 6.5.1 while Section 6.5.2 provides information about how version selection is accomplished in the runtime system.

6.5.1 Compiler Transformations

Four categories of transformations are used in our multiversioning method. Task unrolling, sequentialization and simplification are performed to generate the different versions, and, in a final step, version selection is introduced at each remaining task spawning point in every generated version. All these transformations are applied on INSPIRE.

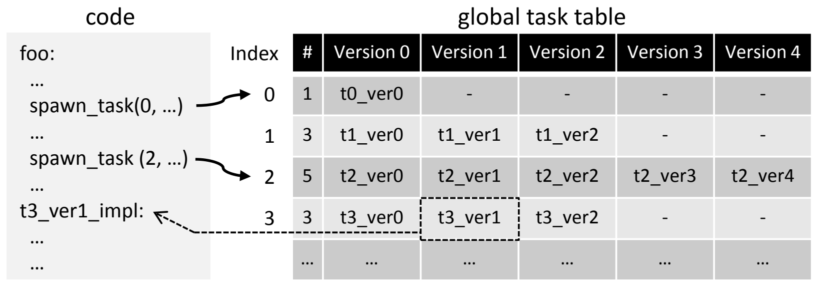
Task Unrolling For each spawn point, this transformation replaces the indirect task invocation with a direct call to the function implementing the task. This function call is then inlined. Simple arguments (e.g. variables) are replaced directly while arguments deduced from complex expressions (e.g. those containing function calls) are stored in newly generated local variables. If there are multiple return statements, control flow structures are adjusted or introduced to maintain the original semantics while replacing the return statements with assignments. The effects of this transformation are illustrated in Figure 6.6 (*b*) and (*c*).

Parameterizing this transformation with an *unrolling factor* N is achieved by repeating the process outlined above N times. The maximum degree of unrolling needs to be controlled in order to prevent an excessive increase in code size, which may impact the effectiveness of the CPU instruction cache and branch prediction unit. This can be accomplished by continuously measuring the code size (in INSPIRE) of the generated versions during compilation and aborting further unrolling and version generation when a threshold value is reached.

Simplification In the simplification step, we apply a number of well known transformations to ease further processing of the code and simplify unnecessarily complex structures that may have been introduced during the previous processing steps. The transformations applied include inlining of very small function calls, constant folding, copy propagation, algebraic simplification, strength reduction and unused code elimination [8].

Version selection Once all the versions are generated, recursive task spawning in each version needs to be adjusted to include the possibility of choosing a different code version at runtime. As described in Section 3.4.3, INSPIRE offers the `pick` primitive to enable multiversioning. All the expressions listed in the `pick` call need to be semantically equivalent, which means that replacing the `pick` primitive by any of its listed expressions is a valid operation maintaining the semantics of the program. The intention is for expressions to be equivalent in their effect on the computational result of the program, but implying distinct performance behavior. Figure 6.7 illustrates a practical application of this primitive in supporting the task multiversioning approach introduced in this chapter.

In the compiler backend, if an instance of `pick` is encountered, code for all versions of the given expressions needs to generated for the corresponding work item. This is accomplished by generating each individual version as a separate function and storing pointers to these functions within a table that is statically generated and embedded within the resulting multiversioned code (Figure 6.8).



Index	#	Version 0	Version 1	Version 2	Version 3	Version 4
0	1	t0_ver0	-	-	-	-
1	3	t1_ver0	t1_ver1	t1_ver2	-	-
2	5	t2_ver0	t2_ver1	t2_ver2	t2_ver3	t2_ver4
3	3	t3_ver0	t3_ver1	t3_ver2	-	-
	...	...	...	...	...	...

Figure 6.8: Task multiversioning implementation in generated code

6.5.2 Runtime System

While the topic of this chapter is an optimization method that could feasibly be applied independently of the task scheduling strategy employed, we will shortly describe the scheduler used in the experiments presented in this chapter. This explanation is intended to make it easier to interpret and compare our results.

The task scheduler we choose to extend with version selection is a *random stealing* based work item scheduler (as described in Section 3.3.3). It uses the low-congestion Insieme-RS *circular work buffer* data structure (see Section 3.5.3) in each worker to keep track of active work item instances.

For this work, we extended this task scheduler. In addition to the circular work buffer, each worker stores an integer value representing current task demand. This value is updated when new work items are generated or a stealing operation fails, as described in Section 6.4.3. Whenever a new instance of a work item is started, the Algorithm 6.2 is evaluated to decide which code version implementing the task should be chosen. This version is then selected from the table generated by the compiler backend (Figure 6.8) and executed.

6.6 Evaluation

In this section we will evaluate the performance impact of our optimization on multiple benchmark programs. Subsection 6.6.1 details our measurement methodology and the experimental setup used. We will perform an in-depth evaluation of two programs in Subsection 6.6.2, and then proceed with an overview of the results of a number of other codes in order to provide a balanced overall impression.

6.6.1 Experimental Setup

For our experiments we used an Intel-based parallel system, incorporating 4 Xeon E7-4870 processors, each comprising 10 physical cores (20 hardware threads) and 3 levels of cache. Table 6.1 summarizes the configuration of this system.

Table 6.1: Hardware and software platform for experimental evaluation.

Sockets/	Cache			Software				
Cores	L1d/i	L2	L3	OS	Kernel	GCC	ICC	Insieme
4/40	32K/32K	256K	30M	CentOS 6.3	2.6.32	4.6.3	12.1	g4614502

When running experiments using a subset of cores, all involved threads were bound to individual physical cores such that the resources of one chip are fully utilized before involving an additional processor. All experimental runs were repeated five times, and the median runtime is reported.

While the most important comparison for our evaluation is between our compiler with and without our multiversioning method, we also included the results obtained by other platforms to provide a reference for comparison. Table 6.1 includes the exact version number of the compilers used in these comparisons. ICC was used as the backend compiler for the Insieme source to source infrastructure, and its built-in Cilk Plus support was employed to compile Cilk programs. The optimization flag "`-O3`" was enabled for all calls to GCC and ICC.

6.6.2 A Detailed Evaluation

N-Queens

The first program we will evaluate is the N-Queens benchmark included in BOTS [31]. Each task in N-Queens spawns 0 to N child tasks, and the depth of its task invocation trees varies from 1 to N, while not following any simple pattern. The size of individual tasks is relatively small.

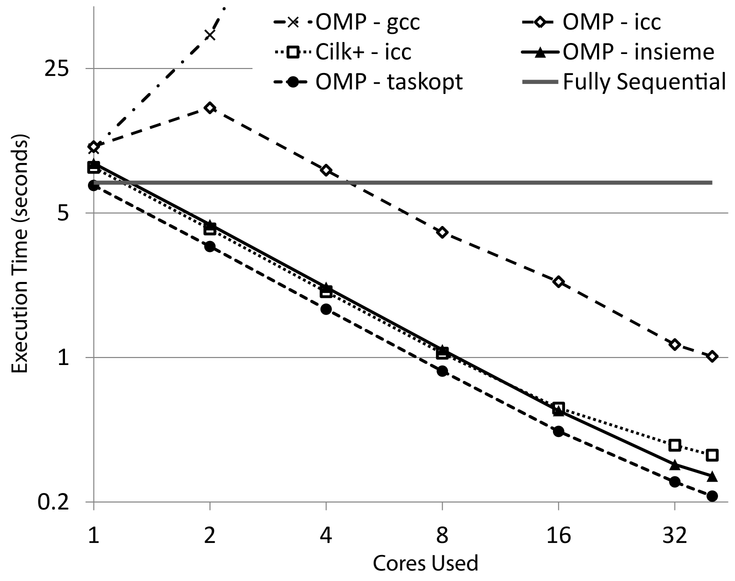


Figure 6.9: N-Queens task benchmark results, $N = 13$, execution time.

Figure 6.9 illustrates the performance of N-Queens using a variety of compilers and implementations. Four OpenMP versions are shown: GCC, ICC and Insieme with ("taskopt") and without ("insieme") task optimization. Additionally, we included the results of a Cilk version and a fully sequential version without any parallel language primitives. The execution time is presented in a log-log plot to improve readability. An efficiency plot is provided in Figure 6.10, which compares the execution times of the parallel versions against ideal scaling from the sequential version.

In terms of OpenMP results, it is clear that the task granularity in this benchmark is too small to be handled effectively by GCC's GOMP implementation. ICC shows the same behavior that was already partially observed in Section 6.2 – execution time increases when going from a single-threaded to a multi-threaded setup. However, starting from two threads performance

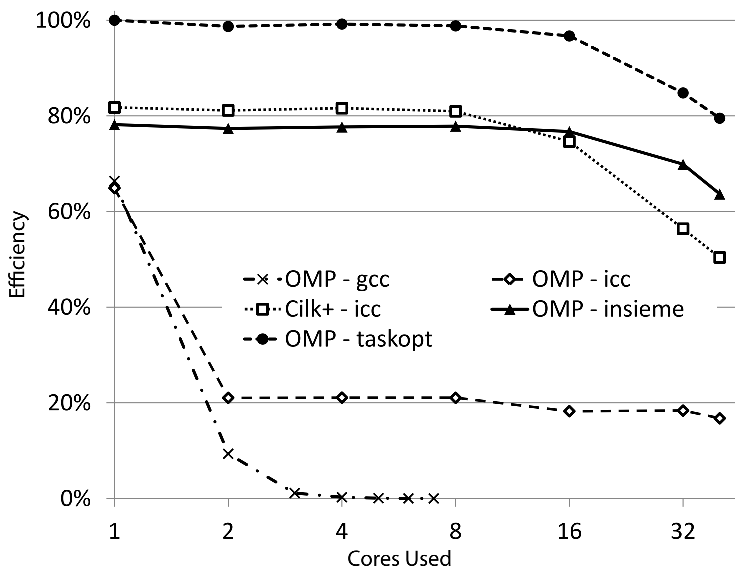


Figure 6.10: N-Queens task benchmark results, $N = 13$, efficiency.

scales relatively well up to 40. Since both of these OpenMP implementations seem ill-equipped to handle very fine-grained tasking well, we also included a Cilk version, which has previously been shown to provide better scaling for fine-grained tasks [63]. Indeed, this implementation performs better in the single-threaded case and scales more smoothly to multiple cores than the GCC and ICC OpenMP versions.

Using Insieme to compile the OpenMP input program results in performance that is comparable to Cilk for up to 16 cores, and scales slightly better beyond this amount. However, a comparison with the fully sequential version indicates that even the Insieme OpenMP version and the Cilk version lose around 20% of performance to overheads incurred due to parallelization. When our task optimization – that is, multiversioning in the compiler and adaptive work item implementation version selection at runtime, as presented in the previous sections – is activated, this overhead is effectively avoided. Even more importantly, this significant reduction in overhead is achieved without negatively affecting the scalability of the program. Performance compared to our implementation without task optimization is improved by 22% to 28% across all measured core counts, with a 25% increase at the full 40 cores.

Compared to the fully sequential version, our approach achieves an efficiency above 99% up to 8 cores, 97% at 16 cores, 85% with 32 cores and 80% at 40 cores. The total runtime of our implementation at higher core counts goes below 0.3 seconds. Note that the drop-off in efficiency primarily occurs at 16 cores and above. This is due to the problem size N=13 causing each initial task to spawn 13 sub-tasks, which means that up to 13 cores can be supplied with work during the first "generation" of tasks. When more cores are used, a larger number of second-generation (and beyond) tasks need to be distributed.

Using the full system (40 cores), our implementation with task optimization improves N-Queens performance by 56% compared to the best competing implementation (Cilk).

Fibonacci

For a second in-depth evaluation, we chose the BOTS Fibonacci program. This is very similar to the code example provided in Section 6.4 (Figure 6.6). As a test case, its most interesting features compared to N-Queens are a significantly different shape of the task invocation tree and the extremely small size of individual tasks. In Fibonacci, each task only creates zero to two sub-tasks, however the maximum depth of the task invocation tree is much larger. Additionally, the depth of the task chains follows an easily predictable pattern, unlike N-Queens.

Note that this is obviously an inefficient method of generating the Fibonacci numbers which would not be used in a production code. However, its properties make it an interesting case study for the overhead of task-parallel systems.

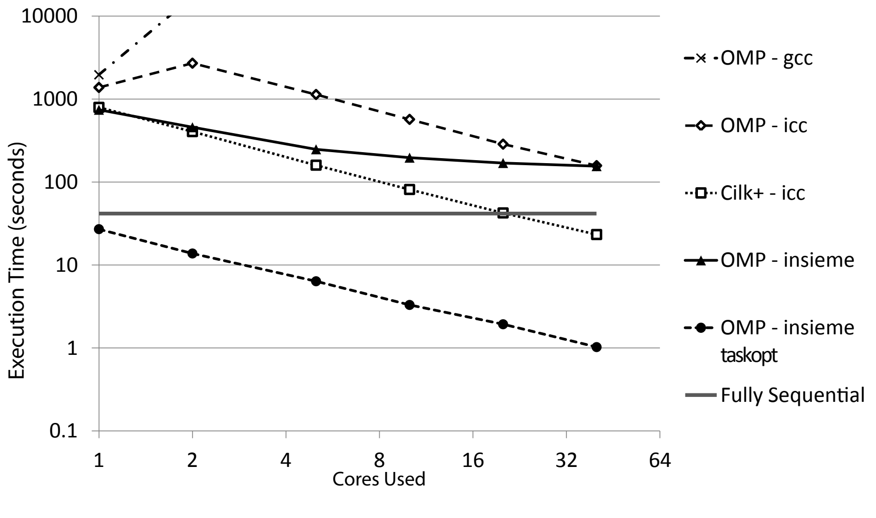


Figure 6.11: Fibonacci benchmark results, $N = 48$.

The performance results achieved in Fibonacci by the set of implementations included in our comparison are illustrated in Figure 6.11, again adjusted to a log-log scale to make them easier to interpret.

The issues with small tasks experienced by the OpenMP implementation in GCC are exacerbated in this case, due to the extremely fine task granularity of the fibonacci program. We stopped the execution of this version after 15000 seconds in the case of 5 or more threads, since these results have no meaningful impact on the comparison. ICC's OpenMP implementation acts similarly to before, with special handling of the single-threaded case and good scaling after the initial parallelism overhead. Cilk is about twice as fast as ICC's OpenMP version in the single-threaded case, and scales well from that point.

As with N-Queens, our Insieme OpenMP implementation without task optimization starts out similarly to Cilk with one and two cores used. However, with a larger number of cores, our scaling behavior suffers. This is due to the very fine-grained nature of the tasks, and the fact that generating a stealable work item induces more overhead in our implementation than it does for Cilk.

However, the most significant result is the performance of the purely sequential version compared to any other existing implementation. Even the most efficient parallel implementations are slowed down by a *factor of 20* when comparing their single-threaded execution time to the fully sequential program. The only existing system that manages to improve on the sequential result at all is Cilk, and it requires 20 cores to do so.

With such small tasks, and therefore relatively large parallelism overheads, it is reasonable to expect that our multiversioning scheme as introduced in this paper will have a large impact on performance. As Figure 6.11 shows, both absolute performance and scalability are greatly improved. With a single thread, runtime is reduced by a *factor of 26* compared to our implementation without multiversioning and adaptive granularity adjustment. Interestingly, the single-threaded execution time using our system is even lower than that of the fully sequential program. This mirrors earlier results in the field of sequential optimization [69], and shows that for very fine-grained tasks, even sequential function call overheads have a relevant impact.

When using all 40 cores of the system, our new approach improves upon the best existing solution (Cilk) by a factor of 23.

6.6.3 Further Benchmarks

Table 6.2 summarizes our benchmark results. It includes measurements for the N-Queens and fib benchmarks presented above, as well as a number of additional programs.

Sort Is the *sort* benchmark included in BOTS.

Strassen Also from BOTS, matrix multiplication using the Strassen algorithm.

Table 6.2: Benchmark Results

Cores:	1	2	5	10	20	40	GM
Queens, $N = 13$ - seq: 7.42							
gcc	10.23	36.29	148.28	308.16	545.22	725.98	
icc	10.49	16.04	6.45	3.81	1.60	0.91	
ins	8.69	4.35	1.74	0.87	0.46	0.27	
opt	6.79	3.41	1.48	0.69	0.36	0.21	
imp	27.92%	27.52%	17.78%	26.64%	25.35%	24.91%	24.75%
Fib, $N = 48$ - seq: 31.09							
gcc	1960.35	17093.63	>15000	>15000	>15000	>15000	
icc	1379.84	2705.65	1135.29	569.15	286.41	157.70	
ins	742.40	456.95	247.91	196.59	169.50	155.29	
opt	27.06	13.77	6.37	3.30	1.93	1.03	
imp	26.43×	32.17×	37.90×	58.15×	86.69×	150.36×	53.92×
Sort, $N = 2^{27}$ - seq: 21.51							
gcc	21.98	11.80	7.20	17.17	29.43	42.29	
icc	23.87	12.36	5.04	2.80	1.85	1.56	
ins	22.94	12.00	4.90	2.71	1.93	1.53	
opt	20.81	11.18	4.61	2.52	1.72	1.41	
imp	5.61%	5.47%	6.43%	7.47%	7.88%	8.11%	6.75%
Strassen, $N = 8192$ - seq: 158.15							
gcc	159.74	92.45	39.20	22.10	15.36	19.94	
icc	164.43	89.94	39.12	21.81	15.69	19.27	
ins	168.84	85.97	37.51	21.98	12.94	8.72	
opt	154.27	79.80	35.46	19.81	12.03	8.11	
imp	3.54%	7.72%	5.77%	10.08%	7.55%	7.52%	6.70%
Stencil, $N = 2048$ - seq: 18.90							
gcc	46.82	62.09	138.51	398.05	576.83	840.61	
icc	30.17	24.65	15.63	14.64	13.84	12.04	
ins	32.49	18.48	9.27	6.31	7.50	9.67	
opt	24.96	13.84	6.66	4.26	5.15	7.54	
imp	20.87%	33.49%	39.17%	47.97%	45.50%	28.29%	34.51%
Floorplan, `input.20` - seq: 17.86							
gcc	27.36	31.04	133.30	352.94	514.51	759.20	
ins	23.53	12.48	5.05	2.53	1.72	1.58	
opt	17.20	9.51	4.12	2.09	1.43	1.24	
imp	36.76%	31.25%	22.62%	21.06%	20.52%	27.68%	26.03%
FFT, $N = 2^{29}$ - seq: 184.78							
gcc	222.27	132.66	95.88	276.81	420.00	482.07	
icc	189.73	112.13	55.95	37.44	22.64	16.03	
ins	187.36	104.85	51.39	36.46	21.01	16.96	
opt	183.97	100.02	49.66	35.08	19.07	12.03	
imp	1.84%	4.84%	3.48%	3.93%	10.16%	33.21%	5.88%
QAP, `chr18a` - seq: 237.28							
gcc	488.97	931.43	7471.11	>15000	>15000	>15000	
icc	785.36	2539.80	823.00	319.87	179.58	114.93	
ins	578.57	294.13	112.80	78.65	70.97	60.71	
opt	231.62	110.76	40.24	21.88	15.18	9.90	
imp	2.11×	2.66×	2.80×	3.59×	4.68×	6.13×	

Stencil A task based 2D stencil computation using the cache-oblivious algorithm presented by Frigo and Strumpen [36]. We included this benchmark to represent an important category of cache-oblivious divide-and-conquer algorithms.

Floorplan The BOTS *floorplan* benchmark. For this application, the binary generated by ICC 12.1 repeatably caused a segmentation fault within ICC's OpenMP library, regardless of the number of threads used. Therefore we are unable to present ICC results for this benchmark.

FFT A parallel fast fourier transform included in BOTS.

QAP A branch and bound solver for quadratic assignment problems.

For every benchmark, the table contains five rows. The results achieved using the GCC and ICC OpenMP implementations are listed in the "gcc" and "icc" rows, respectively. The "ins" row contains the results of our Insieme compiler and runtime without the task multiversioning optimization presented in this paper, while it is enabled for the measurements listed in the "opt" row. Finally, the values in the "imp" row represent the relative improvement achieved using adaptive granularity control, compared to the best result among the other three versions. The columns labeled 1 to 40 correspond to the number of cores used for the computation. All times are given in seconds, and the improvement is provided in percent, except in the case of the Fibonacci and QAP benchmarks where improvement factors are listed instead of very large percentages.

As a frame of reference, the purely sequential time for each benchmark compiled with ICC is provided in each header ("seq"). Note that this time falls between the Insieme time without optimization and the optimized version in most cases, except in the stencil test. Here, the restructuring performed by our compiler prevents some of the low-level sequential optimizations performed by ICC. However, our optimized version executed with one thread is still closer to the sequential performance than any other parallel implementation.

A general trend visible throughout all the benchmark results is the relationship between default task granularity, scaling in GCC and the degree of improvement possible using adaptive task multiversioning and selection. The fibonacci and QAP benchmarks have the most fine grained tasks, and consequently the worst scaling in GCC and the largest improvement with our optimization. On the other end of the spectrum, the FFT, strassen and sort benchmarks feature built-in cutoff values that inherently control task granularity by preventing very small tasks from being generated, resulting in more modest, but still significant, performance improvements with multiversioning. Floorplan, stencil and N-queens fall in between these extremes.

One interesting behavioral pattern which merits some explanation occurs in FFT. Our multiversioning implementation does not result in any significant improvement up to 10 cores, however at 40 cores the measured improvement is 33%. This is due to the FFT benchmark consisting of two separate phases: coefficient calculation and FFT computation. These phases exhibit distinct scaling behaviour, and one of them is affected more significantly by adaptive granularity optimization than the other. Thus, with a larger number of cores, the phase with bad scaling starts to take up a larger portion of the execution time, and the effect of multiversioning on overall performance increases.

6.7 Summary

We have presented a fully automatic, adaptive approach to parallel task granularity control which goes beyond what can be achieved by improving either just a runtime system or focusing only on compilation. By combining a compiler which performs task multiversioning with a runtime system that adaptively selects from these versions, we were able to minimize parallel runtime overhead even for very fine grained tasks.

Our method uses a novel combination of compiler transformations to build an optimized set of semantically equivalent task versions which differ in granularity. The availability of this set of implementations in the compiled program in turn enables our runtime selection algorithm to adjust the amount of tasks generated, while incurring even less overhead than a traditional lazy task creation system with cut-offs.

Evaluating our proposed method across a set of eight benchmarks has shown that our optimization is widely applicable, and that the magnitude of the improvements it enables is related to the task granularity of the input program. For programs with relatively coarse-grained tasks, execution times are reduced by 5% - 10%, while we can achieve improvements of a factor of 6 or more compared to the best competing implementations in fine-grained test cases. Benchmark results also demonstrate that our runtime selection heuristic successfully ensures that scalability (up to 40 cores) is not negatively affected by adaptive task granularity adjustment. Crucially, our adaptive granularity control scheme improves performance in all tested benchmarks and for any given number of cores.

Chapter 7

Conclusion

In this thesis, we have presented Insieme-RS, a parallel runtime system designed to manage multiple types of parallelism – including data parallelism and nested task parallelism – and to effectively map this parallelism to complex, potentially heterogenous, hardware architectures. The main goals of this work are two-fold: improve the performance of parallel codes, generally in terms of execution time but also in other metrics like power efficiency, and at the same time ease the burden on programmers by attempting to automate development and optimization tasks which previously required manual effort. This type of tool support is increasingly necessary as the complexity of parallel hardware architectures continues to increase.

In order to deal with this variety, both in the structure of parallel software as well as in the architecture of target hardware, the Insieme runtime system implements a *multi-paradigm* execution engine for parallel programs. It manages both work and data items in accordance with a novel execution model, and is designed to target shared memory intra-node parallelism, inter-node distributed memory parallelism and accelerator computing. Coarse- and fine-grained task and data parallelism are supported with little overhead, and their scheduling can be dynamically instrumented, steered and optimized. As such, it offers an ideal platform for research in these areas.

As part of the Insieme project, Insieme-RS is closely integrated with the Insieme source-to-source compiler. This symbiotic relationship enables program manipulation and optimization which would be impossible to perform either entirely statically within a compiler, or entirely dynamically in a different runtime system without close compiler integration. Two features enabled by this integration are of particular interest. Firstly, the utilization, with the aid of high-level compiler analysis, of *meta-information* about specific code regions, including symbolic functions which can be evaluated at runtime. Secondly, the generation and encoding of *multiple versions* for some code regions, from which a selection can be made during program execution according to the current state of the system.

The ideas and design of our runtime system are validated by a substantial implementation effort, which, in conjunction with the Insieme compiler, is capable of executing and optimizing a large variety of off-the-shelf parallel programs and benchmarks.

7.1 Contributions

A primary contribution of this thesis is the design of the overall Insieme-RS architecture and its individual components. This includes the specification of a *multi-paradigm execution model for parallel programs with nondeterministic choice*, which provides a unified representation of fork/join, data and task parallelism. It operates on heterogeneous hardware platforms described by means of a detailed *hardware model* designed to capture all information required to support the runtime system in its decision-making processes.

Insieme-RS has been used as a development platform to drive a number of internationally published research efforts, three of which are presented and extended in this thesis. In all cases, the performance of a variety of existing programs is improved over the competitive state of the art by employing techniques made possible by the unique design of our runtime system and its integration with the Insieme compiler.

Multi-Process Scheduling This first individual contribution deals with the automatic scheduling and parallel scaling of multiple OpenMP processes on NUMA machines [83], and is detailed in Chapter 4. The presented technique takes system topology information and per-region scalability metrics into account when making decisions about the degree and mapping of parallelism, and thereby reduces the topological distance between cores executing the same program region.

In a large-scale experiment with a variety of OpenMP benchmarks, we were able to decrease execution time by 33%, compared to conventional methods. Additionally, a 12% reduction in average power consumption was achieved by clustering worker threads on cores belonging to the same socket in the NUMA hierarchy.

Automatic Loop Scheduling The automatic parallel loop scheduling method [82] detailed in Chapter 5 demonstrates the power of a symbiotic relationship between high-level compiler analysis and an intelligent runtime system. For each parallel loop, an effort estimation function is generated at compile time, which maps an iterator range to an estimate for the relative effort or execution time required to execute these iterations. This function is encoded by the compiler backend and evaluated dynamically by Insieme-RS to guide its loop scheduling.

Consequently, the loop scheduling algorithm can take into account the characteristics of the program, as determined by the compiler analysis, as well as dynamic parameters such as the problem size or even external load, which are only available at runtime. In experiments on a set of OpenMP benchmarks, this method achieved improvements of up to 82% in an unloaded system state, and 471% with heavy external load, compared to default OpenMP scheduling.

Task Granularity Control The third individual contribution is a method enabling automatic task granularity control in recursive, task-parallel programs [81], described in Chapter 6. It uses the multi-versioning capabilities of Insieme-RS together with a novel compiler transformation to generate multiple implementation versions of each parallel task with different levels of granularity. During execution, each time a new task is spawned, an algorithm evaluates the current demand for parallelism in the system and selects a suitable task granularity – when the system is saturated, a coarse-grained task version is selected, while fine-grained tasks are preferred when there is a danger of hardware resources idling.

In experimental evaluations, this approach demonstrates good parallel scalability even for very fine-grained tasks. Compared to existing systems, we achieve performance improvements of around 5-10% for programs with relatively course-grained tasks, while improvements of a factor of 6 or more were measured compared to the best competing implementations in fine-grained test cases.

Additional Contributions In addition to the research efforts outlined above, which were driven primarily by Insieme-RS, it also enabled further works as part of the Insieme project. The flexible loop scheduling policies of Insieme-RS were adapted to provide automatic partitioning on heterogenous GPU compute systems [50], while also using meta-data facilities to forward information from the compiler. Furthermore, as part of a multi-objective auto-tuning framework for parallel codes [44], the multi-versioning capabilities of Insieme-RS work items were used to provide statically optimized loop tiling parameters while maintaining the option of dynamically adjusting to changing circumstances during program execution.

7.2 Future Work

Insieme-RS and the whole Insieme project are ambitious efforts, and as such many opportunities for future work and extensions remain. In particular, our combined compiler and runtime approach is quite different from what has been done before in compilers and runtime systems, and requires a re-evaluation of many of the individual works in both fields from this new perspective. Furthermore, due to the large problem space covered by the Insieme-RS design, entire areas of research still remain unexplored.

Distributed Computing One major point of interest is full support for distributed computing, which, although greatly influencing the design and specification of the runtime model, is not yet fully implemented. Here, the knowledge of and ability to manipulate data structures on the compiler side, and being able to forward information about this data layout to the runtime system, could open up new opportunities for automatic or semi-automatic data distribution.

Data Placement The fact that Insieme-RS is aware of the data item dependencies of each work item is not only relevant for fully distributed (cluster) computing. Other potential fields of application include improving data distribution in NUMA systems, or the automatic use of program-managed scratchpad memories in existing GPUs and accelerators as well as upcoming CPU architectures.

Multi-objective Optimization In another area, an even higher level of interactivity between the compiler, the programmer using it and the runtime system could enable new approaches in multi-objective optimization, with the developer stating optimization goals for code regions in the original program, the compiler interpreting and encoding these goals and the runtime system working to implement them. We are currently investigating an annotation-based specification language for multi-objective optimization goals which will be processed by the Insieme compiler and forwarded to Insieme-RS.

Appendices

Table of Symbols

S.	**Semantics**	**P.**
b	A function returning the bandwidth of a connection in the hardware model.	11
$\mathcal{C}$	The control flow graph of a sequential program.	19
$\mathcal{C}^p$	The control flow graph of a parallel program.	22
$\mathcal{C}^n$	The control flow graph of a parallel program with nondeterministic choice.	24
$C_{\mathcal{H}}$	A set of connections between entities in the hardware model.	10
d	A data item.	37
D	The set of all active data items during program execution.	37
E	The set of edges between statements in a program control flow.	19
$E_{\mathcal{H}}$	The set of hardware entities.	10
$e^{\mathcal{H}}$	An entity in the hardware model	10
ϵ	An Insieme-RS event.	49
Γ	The set of parallel join operations in a program.	22
γ	A parallel join operation.	22
g	A communication group.	39
G	The set of all active communication groups during program execution.	39
$\mathcal{H}$	The directed graph representing a target hardware model.	10
k	An Insieme-RS worker.	48
K	The set of all active Workers in an Insieme-RS process.	48
l	A function returning the latency of a connection in the hardware model.	11
$\mathbf{m}$	A partial function to query meta-information.	25

Table 1: Table of symbols (1).

S.	**Semantics**	**P.**
$\mathcal{P}$	A sequential program.	19
$\mathcal{P}^p$	A parallel program.	22
$\mathcal{P}^n$	A parallel program with nondeterministic choice.	24
Ψ	The set of parallel spawn operations in a program.	22
ψ	A parallel spawn operation.	22
q	The resource requirement function of a work item.	26
R	A single-exit single-entry program region	20
$\mathbf{r}$	A function capturing read accesses to a variable.	20
S	The set of statements in a program.	19
S^p	A set of program statements and parallel operations.	22
S^n	A set of program statements, parallel and choice operations.	24
s	A program statement.	19
$\hat{s}$	The execution state of a program.	28
$\mathbf{S}_n$	The parallel speedup of a program with n-fold parallelism.	22
$T_{\mathcal{H}}$	A type identifier in hardware model addressing.	17
τ	The type of a program variable.	20
Θ	The set of nondeterministic choice operations in a program.	24
θ	A nondeterministic choice operation.	24
V	A set of variables.	19
$\hat{V}$	A set of variable assignments in a program state.	28
$\mathbf{w}$	A function capturing write accesses to a variable.	20
$\overline{w}$	A work item description.	25
$\overline{W}$	The set of all work item descriptions.	24
w	A work item instance.	31
W	The set of all active work item instances.	30
X	The set of parallel communication operations in a program.	22
χ	A parallel communication operation.	22

Table 2: Table of symbols (2).

List of Figures

List of Tables

List of Definitions

List of Listings

Bibliography

[1] Andrei Alexandrescu. Lock-free data structures. *C/C++ User Journal*, 2004.

[2] Apple. Grand central dispatch technology brief. Whitepaper, 2009.

[3] ARM. Arm neon general-purpose simd engine. 2010.

[4] Asanovic and Krste et al. The landscape of parallel computing research: A view from berkeley. Technical report, EECS Department, University of California, Berkeley, 2006.

[5] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[6] E. Ayguad, B. Blainey, Duran, Labarta A., Martnez J., Martorell F., and R. X., Silvera. Is the schedule clause really necessary in openmp? Lecture Notes in Computer Science on OpenMP Shared Memory Parallel Programming (2003).

[7] Eduard Ayguadé, Rosa M Badia, Francisco D Igual, Jesús Labarta, Rafael Mayo, and Enrique S Quintana-Ortí. An extension of the starss programming model for platforms with multiple gpus. In *Euro-Par 2009 Parallel Processing*, pages 851–862. Springer, 2009.

[8] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994.

[9] Lasinski T. Simon H. Bailey D., Barton J. The nas parallel benchmarks. 1991.

[10] Muthu Manikandan Baskaran, Nagavijayalakshmi Vydyanathan, Uday Kumar Reddy Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective par-

allelization of loop nests on multicore processors. *SIGPLAN Not.*, 44(4):219–228, February 2009.

[11] C. Bastoul. Improving data locality in static control programs. PhD thesis, University Paris 6, Pierre et Marie Curie, France (2004).

[12] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott. ompverify: polyhedral analysis for the openmp programmer. In *Proceedings of the 7th international conference on OpenMP in the Petascale era*, IWOMP'11, pages 37–53, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] Pieter Bellens, Josep M Perez, Rosa M Badia, and Jesus Labarta. Cellss: a programming model for the cell be architecture. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 5–5. IEEE, 2006.

[14] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303, Berlin, Heidelberg, 2010. Springer-Verlag.

[15] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proc. 5th ACM SIGPLAN symp. on Principles and practice of parallel programming*, PPOPP '95, pages 207–216, 1995.

[16] OpenMP Architecture Review Board. Openmp application program interface, version 3.0. 2008.

[17] Uday Bondhugula and J. Ramanujam. Pluto: A practical and fully automatic polyhedral program optimization system. In *In Proceedings of the ACM SIGPLAN 2008 conference on programming language design and implementation (PLDI 08)*, 2008.

[18] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in heterogeneous computing. *Sci. Program.*, 18(1):1–33, January 2010.

[19] François Broquedis, Thierry Gautier, and Vincent Danjean. Libkomp, an efficient openmp runtime system for both fork-join and data flow paradigms. In *Proc. 8th int. conf. on OpenMP in a Heterogeneous World*, IWOMP'12, pages 102–115, 2012.

[20] Mike Butler. Amd bulldozer core-a new approach to multithreaded compute performance for maximum efficiency and throughput. In *IEEE HotChips Symposium on High-Performance Chips (HotChips 2010)*, 2010.

[21] Franck Cappello and Olivier Richard. Intra node parallelization of mpi programs with openmp. Technical report, 1998.

[22] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[23] Barbara Chapman, Lei Huang, C. Bischof, M. Bcker, P. Gibbon, G. R. Joubert, B. Mohr, F. Peters (eds, Barbara Chapman, and Lei Huang. Enhancing openmp and its implementation for programming multicore systems. Advances in Parallel Computing, Volume 15, IOS Press (2008), 2008.

[24] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, volume 40, pages 519–538. ACM, 2005.

[25] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: a performance view. *IBM J. Res. Dev.*, 51(5):559–572, September 2007.

[26] Xuan Chen and Shun Long. Adaptive multi-versioning for openmp parallelization via machine learning. In *Proc. 15th Int. Conf. on Parallel and Distributed Systems*, ICPADS '09, pages 907–912, 2009.

[27] Neil Ashish Deshpande and Stephen A. Edwards. Statically unrolling recursion to improve opportunities for parallelism. Technical report, Department of Computer Science, Columbia University, 2012.

[28] The Insieme development team. Insieme compiler and runtime infrastructure. http://insieme-compiler.org.

[29] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. An adaptive cut-off for task parallelism. In *Proc. 2008 ACM/IEEE conf. on Supercomputing*, SC '08, pages 36:1–36:11, 2008.

[30] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. Evaluation of openmp task scheduling strategies. In *Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, IWOMP'08, pages 100–110, Berlin, Heidelberg, 2008. Springer-Verlag.

[31] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Proc. 2009 Int. Conf. on Parallel Processing*, ICPP '09, pages 124–131, 2009.

[32] David N. Turner (ed), Hans Wolfgang Loidl, and Kevin Hammond. On the granularity of divide-and-conquer parallelism. In *Glasgow Workshop on Functional Programming*, pages 8–10. Springer-Verlag, 1995.

[33] Karl filip Faxn (editor, Christer Bengtsson, Mats Brorsson, Erik Hagersten, Bengt Jonsson, Christoph Kessler, Bjrn Lisper, Per Stenstrm, and Bertil Svensson. Multicore computing – the state of the art. `http://eprints.sics.se/3546/`, 2008.

[34] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel avx: New frontiers in performance improvements and energy efficiency. *Intel white paper*, 2008.

[35] Stephen Fitzpatrick, M. Clint, and P. Kilpatrick. Unfolding recursive function definitions using the paradoxical combinator, 1996.

[36] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *Proc. 19th int. conf. on Supercomputing*, ICS '05, pages 361–366, 2005.

[37] Thierry Gautier, Fabien Lementec, Vincent Faucher, and Bruno Raffin. X-Kaapi: a Multi Paradigm Runtime for Multicore Architectures. Rapport de recherche RR-8058, INRIA, February 2012.

[38] Sebastian Herbert. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *in International Symposium on Low Power Electronics and Design*, 2007.

[39] Intel. Sse programming reference. Intel software network, 2007.

[40] Intel. Intel c and c++ compilers. http://software.intel.com/en-us/c-compilers/, 2012.

[41] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

[42] H. Jordan, P. Thoman, J.J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch. A multi-objective auto-tuning framework for parallel codes. In *Proc. 2012 ACM/IEEE Int. Conf. on Supercomputing*, page to appear, 2012.

[43] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. Inspire the insieme parallel intermediate representation. In *PACT '13: Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. ACM, 2013.

[44] Herbert Jordan, Peter Thoman, Juan J. Durillo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. A multi-objective auto-tuning framework for parallel codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 10:1–10:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[45] Herbert Jordan, Peter Thoman, and Thomas Fahringer. A high-level ir transformation system. In *Euro-Par 2013: Parallel Processing Workshops*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013.

[46] Norman P Jouppi and David W Wall. *Available instruction-level parallelism for superscalar and superpipelined machines*, volume 17. ACM, 1989.

[47] Laxmikant V Kale and Sanjeev Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[48] A. Kleen. A numa api for linux. `http://halobates.de/numaapi3.pdf`, 2004.

[49] B. Knafla and C. Leopold. Parallelizing a real-time steering simulation for computer games with openmp. Proc. Parallel Computing (ParCo), pages 219226., 2007.

[50] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pages 149–160, New York, NY, USA, 2013. ACM.

[51] Graud Krawezik and Cappello F. Performance comparison of mpi and three openmp programming styles on shared memory multiprocessors. In *In ACM SPAA 2003*, pages 118–127. ACM Press, 2003.

[52] Jesus Labarta. Starss: A programming model for the multicore era. In *PRACE WorkshopNew Languages & Future Technology Prototypes at the Leibniz Supercomputing Centre in Garching (Germany)*, 2010.

[53] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Acm Sigplan Notices*, volume 44, pages 227–242. ACM, 2009.

[54] Tong Li, Dan Baumberger, and Scott Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 65–74, New York, NY, USA, 2009. ACM.

[55] David B Loveman. High performance fortran. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 1(1):25–42, 1993.

[56] Maged M Michael. Scalable lock-free dynamic memory allocation. In *ACM SIGPLAN Notices*, volume 39, pages 35–46. ACM, 2004.

[57] Eric Mohr, David A. Kranz, Robert H. Halstead, and Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2, 1991.

[58] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing.* O'Reilly Media, Inc., 1996.

[59] R. Noronha and D.K Panda. Improving scalability of openmp applications on multi-core systems using large page support. Parallel and Distributed Processing Symp., IPDPS 2007. IEEE International 26-30 March 2007.

[60] Diego Novillo. Openmp and automatic parallelization in gcc. In *In the Proceedings of the GCC Developers, `http://gcc.gnu.org/projects/gomp/`*, 2006.

[61] NVidia. Compute unified device architecture (cuda) programming guide. 2007.

[62] NVidia. Nvidias next generation cuda compute architecture: Kepler gk110. 2012.

[63] Stephen Olivier and Jan F. Prins. Comparison of openmp 3.0 and other task parallel frameworks on unbalanced task graphs. *International Journal of Parallel Programming*, 38(5-6):341–360, 2010.

[64] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. Openmp task scheduling strategies for multicore numa systems. *Int. J. High Perform. Comput. Appl.*, 26(2):110–124, May 2012.

[65] OpenMP Architecture Review Board. OpenMP Specification. Version 3.1, July 2011.

[66] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[67] R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436, 2009.

[68] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* O'Reilly Media, Inc., 2010.

[69] Radu Rugina and Martin C. Rinard. Recursion unrolling for divide and conquer programs. In *Proc. 13th Int. Workshop on Languages and Compilers for Parallel Computing*, LCPC '00, pages 34–48, 2001.

[70] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):18:1–18:15, August 2008.

[71] Michael D. Smith, Monica S. Lam, and Mark A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 344–354, New York, NY, USA, 1990. ACM.

[72] Marc Snir, Steve W Otto, David W Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: the complete reference.* MIT press, 1995.

[73] F. Somenzi. Cudd: Cu decision diagram package.

[74] Richard Stallman. Using and porting the gnu compiler collection. *M.I.T. Artificial Intelligence Laboratory*, 2001.

[75] Richard W. Stevens and Stephen A. Rago. *Advanced Programming in the UNIX(R) Environment (2nd Edition).* Addison-Wesley Professional, 2005.

[76] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[77] Inc. Sun Microsystems. Sun fire x4600 m2 server architecture. White Paper, `http://www.sun.com/servers/x64/x4600/arch-wp.pdf`, 2008.

[78] Håkan Sundell and Philippas Tsigas. Lock-free deques and doubly linked lists. *Journal of Parallel and Distributed Computing*, 68(7):1008–1020, 2008.

[79] Gerald Jay Sussman and Guy L Steele Jr. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.

[80] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.

[81] Peter Thoman, Herbert Jordan, and Thomas Fahringer. Adaptive granularity control in task parallel programs using multiversioning. In *Euro-Par 2013 - Parallel Processing, 19h International Euro-Par Conference, Aachen, Germany, Proceedings*, Lecture Notes in Computer Science. Springer, 2013.

[82] Peter Thoman, Herbert Jordan, Simone Pellegrini, and Thomas Fahringer. Automatic openmp loop scheduling: a combined compiler and runtime approach. In *Proc. 8th int. conf. on OpenMP in a Heterogeneous World*, IWOMP'12, pages 88–101, 2012.

[83] Peter Thoman, Hans Moritsch, and Thomas Fahringer. Topology-aware openmp process scheduling. In Mitsuhisa Sato, Toshihiro Hanawa, MatthiasS. Mller, BarbaraM. Chapman, and BronisR. Supinski, editors, *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, volume 6132 of *Lecture Notes in Computer Science*, pages 96–108. Springer Berlin Heidelberg, 2010.

[84] K. Trifunovic and A. Cohen. Graphite two years after: First lessons learned from real-world polyhedral compilation. GCC Research Opportunities Workshop (GROW) (2010).

[85] Alexandros Tzannes, George C Caragea, Rajeev Barua, and Uzi Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *ACM Sigplan Notices*, volume 45, pages 179–190. ACM, 2010.

[86] T. Tzen, T.H. Tzen, L. Ni, and L.M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. IEEE Transactions onParallel and Distributed Systems (1993).

[87] S. Verdoolaege. barvinok: User guide. `http://www.kotnet.org/~skimo/barvinok/barvinok.pdf`.

[88] Z. Wang and M. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP) (2009).

[89] Maurice V. Wilkes. The memory gap and the future of high performance memories. *SIGARCH Comput. Archit. News*, 29(1):2–7, March 2001.

[90] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32. ACM, 2007.

[91] Y. Zhang, M. Burcea, V. Cheng, Ho R., and M. Voss. An adaptive openmp loop scheduler for hyperthreaded smps. Proc. of PDCS-2004: International Conference on Parallel and Distributed Computing Systems (2004).