UNIVERSITY OF CALIFORNIA,
IRVINE

Optimizing Many-Threads-to-Many-Cores Mapping in Parallel Electronic System Level Simulation

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Guantao Liu

2017

DEDICATION

To my dearest parents
for their unconditional love and support

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

It has been a long and wonderful journey to pursue the doctorate degree during the past five years, and life would have been much more difficult without the support from my professors, family, colleagues and friends. Thus, I would like to take this opportunity to thank these great people for encouraging and inspiring me in this journey.

First and foremost, I would like to gratefully and sincerely thank my advisor, Professor Rainer Dömer, for his guidance, motivation, understanding and patience during my graduate study at UC Irvine. It has been a great learning experience to be a member of his research group and a teaching assistant of his class. His considerable insights, kind personality and sense of humor created an enjoyable and pleasant working environment in our group. Thanks to this, I have learned much more from our meetings and conversations in the past few years, not only on researching and teaching, but also on life and philosophy. This is a treasure that will benefit me for a whole lifetime.

Next, I would like to thank Professor Kwei-Jay Lin and Professor Mohammad A. Al Faruque, for taking time out of their busy schedules to serve as my committee members and provide me constructive suggestions and feedback on my dissertation.

Many thanks to my colleagues in the LECS group, including Weiwei Chen, Xu Han, Yasaman Samei, Che-Wei Chang and especially Tim Schmidt, for the fruitful talks, discussions and teamwork. I appreciate all their help during my graduate program.

Also, I am thankful to all my friends and colleagues in the Chinese Students and Scholars Association at UC Irvine. Those wonderful events and activities we attended and hosted together have greatly enriched my nerdy social life, and they always gave me joy and happiness to continue my research in the United States.

This work has been supported in part by funding from Intel Corporation. I thank Intel for the valuable support.

Last but not least, I would like to express my deepest gratitude to my parents, Xinyong Liu and Biyuan She, for their unconditional love and endless faith in me. Their support is the very source of power and morale for me to pursue my dream.

CURRICULUM VITAE

Guantao Liu

EDUCATION

Doctor of Philosophy in Computer Engineering	**2017**
University of California, Irvine	*Irvine, California*
Master of Science in Electrical and Computer Engineering	**2013**
University of California, Irvine	*Irvine, California*
Bachelor of Engineering in Information Engineering	**2011**
Southeast University	*Nanjing, China*

RESEARCH EXPERIENCE

Graduate Student Researcher	**2011–2017**
University of California, Irvine	*Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant	**2015–2016**
University of California, Irvine	*Irvine, California*

INDUSTRIAL EXPERIENCE

Software Engineer Intern	**2016**
Arista Networks	*Santa Clara, California*
Systems Software Intern	**2014**
Samsung Research America - Sillicon Valley	*San Jose, California*
IC Design Verification Intern	**2012–2013**
Broadcom	*Irvine, California*

REFEREED JOURNAL PUBLICATIONS

G. Liu, T. Schmidt, and R. Dömer. **A Communication-Aware Thread Mapping Framework for SystemC PDES**. In preparation, February 2017.

W. Chen, X. Han, C. Chang, G. Liu, and R. Dömer. **Out-of-Order Parallel Discrete Event Simulation for Transaction Level Models**. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 12, pp. 1859-1872, December 2014.

REFEREED BOOK CHAPTERS

R. Dömer, G. Liu, and T. Schmidt. **Parallel Simulation**. Chapter 18 in *Handbook of Hardware/Software Codesign* by S. Ha and J. Teich, Springer, accepted for publication, August 2016.

REFEREED CONFERENCE PUBLICATIONS

T. Schmidt, G. Liu, and R. Dömer. **Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation**. In *Proceedings of Design Automation Conference 2017*, accepted for publication, February 2017.

T. Schmidt, G. Liu, and R. Dömer. **Hybrid Analysis of SystemC Models for Fast and Accurate Parallel Simulation**. In *Proceedings of the Asia and South Pacific Design Automation Conference 2017*, Tokyo, Japan, January 2017.

G. Liu, T. Schmidt, and R. Dömer. **A Segment-Aware Multi-Core Scheduler for SystemC PDES**. In *Proceedings of the International High Level Design Validation and Test Workshop 2016*, Santa Cruz, California, October 2016.

T. Schmidt, G. Liu, and R. Dömer. **Automatic Generation of Thread Communication Graphs from SystemC Source Code**. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, St. Goar, Germany, May 2016.

G. Liu, T. Schmidt, and R. Dömer. **Optimizing Thread-to-Core Mapping on Manycore Platforms with Distributed Tag Directories**. In *Proceedings of the Asia and South Pacific Design Automation Conference 2015*, Tokyo, Japan, January 2015.

MASTER THESIS

G. Liu. **A Hybrid Thread Library for Efficient Electronic System Level Simulation**. Master's thesis, University of California, Irvine, Irvine, California, June 2013.

TECHNICAL REPORTS

G. Liu, T. Schmidt, and R. Dömer. **RISC Compiler and Simulator, Beta Release V0.3.0: Out-of-Order Parallel Simulatable SystemC Subset**. Center for Embedded and Cyber-Physical Systems, Technical Report 16-06, September 2016.

G. Liu, T. Schmidt, and R. Dömer. **RISC Compiler and Simulator, Alpha Release V0.2.1: Out-of-Order Parallel Simulatable SystemC Subset**. Center for Embedded and Cyber-Physical Systems, Technical Report 15-02, October 2015.

G. Liu, and R. Dömer. **A Hybrid Thread Library for Efficient Electronic System Level Simulation**. Center for Embedded Computer Systems, Technical Report 13-11, October 2013.

G. Liu, and R. Dömer. **A User-Level Thread Library Built on Linux Context Functions for Efficient ESL Simulation**. Center for Embedded Computer Systems, Technical Report 13-07, June 2013.

G. Liu, and R. Dömer. **Performance Evaluation and Optimization of A Custom Native Linux Threads Library**. Center for Embedded Computer Systems, Technical Report 12-11, October 2012.

OPEN SOURCE SOFTWARE RELEASES

R. Dömer, G. Liu, and T. Schmidt. **RISC Compiler and Simulator, Beta Release V0.3.0**. `http://cecs.uci.edu/~doemer/risc.html#RISC030`, September 2016.

R. Dömer, G. Liu, and T. Schmidt. **RISC Compiler and Simulator, Alpha Release V0.2.1**. `http://cecs.uci.edu/~doemer/risc.html#RISC021`, October 2015.

R. Dömer, G. Liu, and T. Schmidt. **RISC Compiler and Simulator, Alpha Release V0.2.0**. `http://cecs.uci.edu/~doemer/risc.html#RISC020`, September 2015.

R. Dömer, G. Liu, and T. Schmidt. **RISC API, Alpha Release V0.1.0**.
`http://cecs.uci.edu/~doemer/risc.html#RISC010`, June 2014.

ABSTRACT OF THE DISSERTATION

Optimizing Many-Threads-to-Many-Cores Mapping
in Parallel Electronic System Level Simulation

By

Guantao Liu

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2017

Professor Rainer Dömer, Chair

In hardware/software codesign, Discrete Event Simulation (DES) has been in use for decades to verify and validate the functionality of Electronic System Level (ESL) models. Since the parallel computing platforms are readily available today, many Parallel Discrete Event Simulation (PDES) approaches are proposed to improve the simulation performance. However, as the thread parallelism increases in ESL designs and core count multiplies on multi-core and many-core platforms, thread-to-core mapping becomes critical in PDES.

In this dissertation, we propose a *computation- and communication-aware* approach to optimize thread mapping for parallel ESL simulation, with the aims of load balancing and communication minimization. As we identify that the order of dispatching parallel threads has a significant influence on the total simulation time, and Longest Job First (LJF) shows better performance than the Linux default thread dispatch policy, we first propose a *segment-aware* LJF scheduler for PDES. Our segment-aware scheduler can accurately predict the run time of the thread segments ahead, and thus make better dispatching decisions. Next, we define the concept of *core distance* for multi-core and many-core architectures, which quantifies core-to-core communication latency and characterizes processor hierarchies. For many-core architectures using directory-based cache coherence protocols, we observe that core-to-core transfers are not always significantly faster than main memory accesses, and the core-to-core communication latency depends not only on the physical placement on the chip, but also on the location of the distributed cache tag directory. Thus, using a ping-pong memory benchmark, we quantify the core distance on a ring-network many-core platform and propose an algorithm to optimize thread-to-core mapping in order to minimize on-chip communication overhead. Altogether, based on a static analysis of communication patterns and core distance and a dynamic profiling of computation load, our proposed framework utilizes a heuristic graph partitioning algorithm and automatically generates an optimized thread mapping, which minimizes inter-chip communication overhead. In our systematic evaluation, our approach consistently shows a significant performance gain on top of the order-of-magnitude speedup of PDES.

The contributions of this dissertation include a segment-aware multi-core scheduler, core distance profiling, a communication-aware thread mapping framework, together with an open-source software package for Out-of-Order PDES.

Chapter 1

Introduction

Embedded and cyber-physical systems are pervasive and ubiquitous nowadays, covering automotive and avionic systems, medical devices, smart home appliances, mobile and consumer electronics, and others. However, due to the increasing size, complexity and heterogeneity of the embedded and cyber-physical systems, it is extremely difficult for system designers to consider all the implementation details at the early stages of the design [72, 62]. Therefore, Electronic System Level (ESL) [62] design methodology is proposed to cope with such design challenges. With this approach, designers can elevate the abstraction level of the design and hide low-level implementation details, and thus focus on the functional specification and algorithms in the design. Before the system level design is refined to a lower abstraction level, it must be verified to assure the correctness [28]. The most common approach to verify the system level design is *Simulation-based Validation*, which is relatively fast yet accurate [25]. As the parallel processing capabilities are readily available in today's multi-core and many-core hosts, *parallel* ESL simulation has recently attracted a lot of attention. In this dissertation, we aim at optimizing the thread-to-core mapping in parallel Electronic System Level simulation.

1.1 System Level Design and Simulation

The 2004 edition of the International Technology Roadmap for Semiconductors defines system level as *"a level above RTL including both hardware and software design"* [41]. In order to address the design challenges of radically increasing size and complexity of both embedded hardware and software, system level design is proposed as a holistic approach to cover the complete picture of the entire system. Figure 1.1 illustrates the complexities of different levels of abstraction in system design. Clearly, while there are tens of millions of transistors in a system, the number of components in the system level is reduced to less than ten, which mainly consist of hardware platforms and software implementations. The trade-off here is that the higher the abstraction level is, the less complexity and accuracy the model has. Due to the design challenges of embedded systems nowadays, *"a well-known solution for dealing with complexity is to exploit hierarchy and to move to higher levels of abstraction"* [24]. Thus, system level modeling is a promising approach at the early stages of the design.

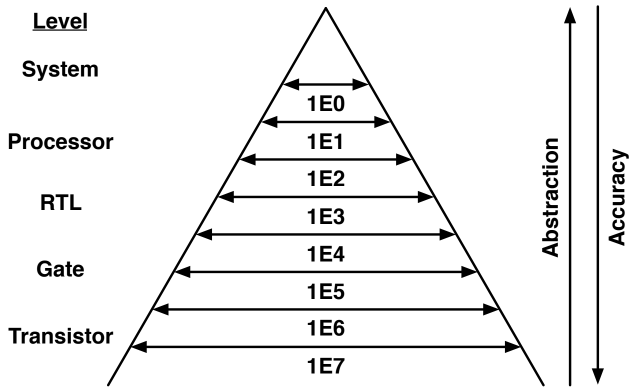


Figure 1.1: Level of abstraction in system design [31].

In order to shorten the time-to-market design period, fast and accurate ESL simulation is critical in system design and validation. Basically, there are three types of computer-based simulation: discrete event, continuous and Monte Carlo [65]. As digital systems are naturally discrete, most of the ESL simulation approaches use discrete event semantics. Next, we discuss different Discrete Event Simulation (DES) scheduling algorithms.

1.1.1 Discrete Event Simulation

In order to describe hardware and software components in system level designs, System-Level Description Languages (SLDLs) are proposed to add supports of system level modeling, such as behavioral and structural hierarchy, concurrency, communication, synchronization, and timing constraints [25]. Two predominant examples of SLDLs are SystemC [33] and SpecC [29]. While SpecC is a superset of ANSI-C and widely used in academia, SystemC is a C++ library and the de-facto language for system level design in industry. Also, SystemC is published as an IEEE standard for system design [37]. Thus, we use SystemC as the example of SLDL in this dissertation.

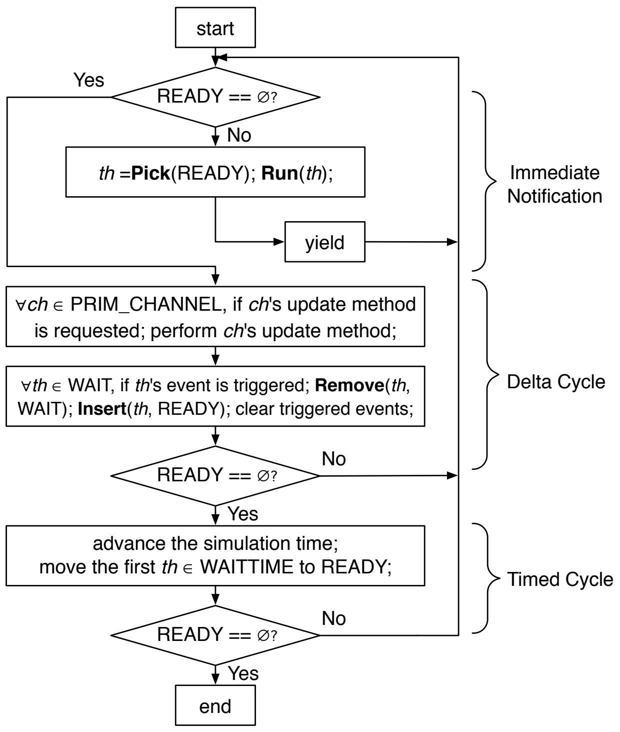

start
READY == ∅?
Yes
No
th =Pick(READY); Run(th);
yield
Immediate Notification
∀ch ∈ PRIM_CHANNEL, if ch's update method is requested; perform ch's update method;
∀th ∈ WAIT, if th's event is triggered; Remove(th, WAIT); Insert(th, READY); clear triggered events;
Delta Cycle
READY == ∅?
No
Yes
advance the simulation time; move the first th ∈ WAITTIME to READY;
Timed Cycle
READY == ∅?
No
Yes
end

Figure 1.2: Traditional Discrete Event Simulation (DES) scheduler for SystemC [56].

Figure 1.2 depicts a traditional Discrete Event Simulation (DES) scheduling algorithm for SystemC. In DES, we have the following definitions of data structures and operations [26]:

1. Definition of thread queues in DES:

- $\mathsf{QUEUES} = \{\mathsf{READY}, \mathsf{RUN}, \mathsf{WAIT}, \mathsf{WAITTIME}, \mathsf{COMPLETE}\}$.
- $\mathsf{READY} = \{th \mid th \text{ is ready to run}\}$.
- $\mathsf{RUN} = \{th \mid th \text{ is currently running}\}$.
- $\mathsf{WAIT} = \{th \mid th \text{ is waiting for an event}\}$.
- $\mathsf{WAITTIME} = \{th \mid th \text{ is waiting for time advance}\}$.
- $\mathsf{COMPLETE} = \{th \mid th \text{ has completed its execution}\}$.

2. Simulation invariants in DES:

   Let $\mathsf{THREADS}$ be the set of all threads created in DES. Then, at all times, the following conditions hold:

   - $\mathsf{THREADS} = \mathsf{READY} \cup \mathsf{RUN} \cup \mathsf{WAIT} \cup \mathsf{WAITTIME} \cup \mathsf{COMPLETE}$.
   - $\forall A, B \in \mathsf{QUEUES}, A \neq B : A \cap B = \varnothing$.

3. Operations on threads and queues:

   - **Run**(th): Dispatch thread th and th starts execution.
   - $th =$ **Pick**(READY): Pick a thread th from the READY queue.
   - **Remove**(th, WAIT): Remove the thread th from the WAIT queue.
   - **Insert**(th, READY): Add the thread th to the READY queue.

4. Initial state at the beginning of DES:

   - $\mathsf{THREADS} = \{th_{root}\}$.
   - $\mathsf{RUN} = \{th_{root}\}$.
   - $\mathsf{READY} = \mathsf{WAIT} = \mathsf{WAITTIME} = \mathsf{COMPLETE} = \varnothing$.
   - $(t, \delta) = (0, 0)$ where t represents the timed cycle, and δ represents the delta cycle.

In DES, the timed cycle represents the acutal time advance, and the delta cycle interprets the zero-delay semantics in digital systems. Specifically, the delta cycle lasts for an infinitesimal amount of time and imposes a partial order of simulation actions [33]. Also, SystemC standard allows to notify an event immediately within the same delta cycle [37]. Thus, the scheduling algorithm in Figure 1.2 implies three causal loops resulting from immediate notification, delta cycle and timed cycle.

In Figure 1.2, it is clear that there is a single thread running at all times. When one thread finishes its current evaluation phase, it yields to another thread in the READY queue. When all threads in the READY queue complete their current delta cycle, the root thread resumes and performs the channel update and event notification. Then, a new delta cycle begins. If no more threads are runnable after the update and notification, the scheduler advances the simulation time and starts a new timed cycle. The earliest timed event is processed and the associate thread is moved from the WAITTIME queue to the READY queue. The traditional DES terminates when both the WAITTIME and READY queues are empty.

1.1.2 Synchronous Parallel Discrete Event Simulation

In comparison to DES, a Parallel Discrete Event Simulation (PDES) [27] scheduler dispatches multiple threads from the READY queue concurrently. Figure 1.3 shows a synchronous PDES scheduler for SystemC.

In the synchronous PDES scheduling algorithm, as long as the READY queue is not empty and an idle core is available, one more thread is dispatched from the READY queue. When a thread finishes earlier than other threads in the same delta cycle, a new ready thread is assigned to the idle core, unless no more thread is available in the READY queue. In this case, the processing core keeps idle until the next delta cycle or any event is immediately notified. Then, after all threads finish their current delta cycle, the last running thread performs the

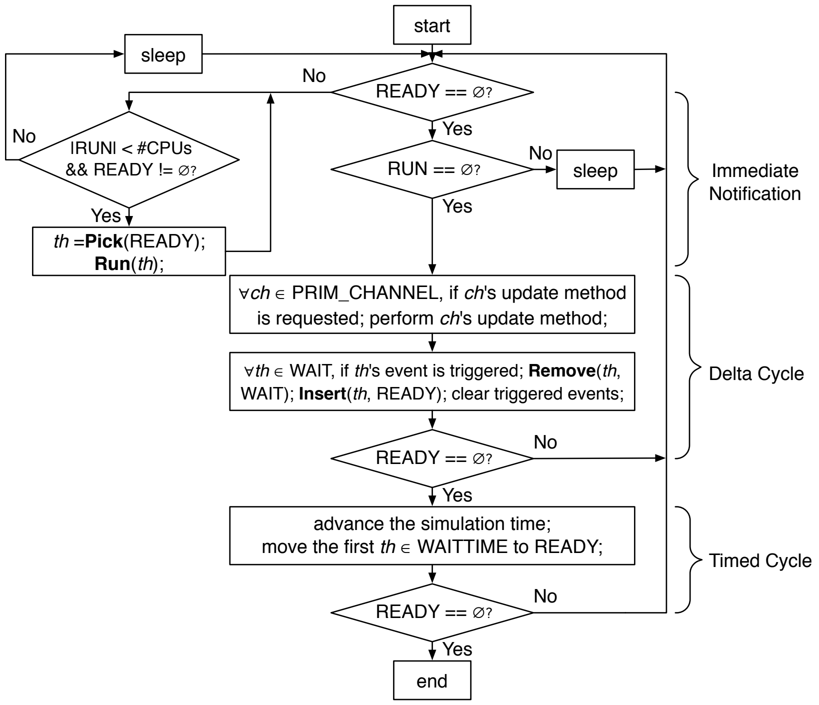

start
sleep
No
READY == ∅?
Yes
No
|RUN| < #CPUs && READY != ∅?
Yes
th =Pick(READY); Run(th);
RUN == ∅?
No
sleep
Yes
Immediate Notification
∀ch ∈ PRIM_CHANNEL, if ch's update method is requested; perform ch's update method;
∀th ∈ WAIT, if th's event is triggered; Remove(th, WAIT); Insert(th, READY); clear triggered events;
Delta Cycle
READY == ∅?
No
Yes
advance the simulation time; move the first th ∈ WAITTIME to READY;
Timed Cycle
READY == ∅?
No
Yes
end

Figure 1.3: Synchronous Parallel Discrete Event Simulation (PDES) scheduler for SystemC [56].

update and notification phases, and advances the timed cycle if the READY queue is still empty. Note that the synchronous PDES implies an absolute barrier at the end of delta and timed cycle. All threads are blocked at the barrier until any other threads finish their current evaluation phase.

In order to avoid race conditions among accesses to internal scheduling resources, synchronous PDES introduces a mutex lock to protect the SystemC kernel. Any thread needs to acquire the mutex before modifying the state of the kernel. For safe communication, the channel between modules is explicitly protected with a local lock. Also, note that while any threading model (user-level or kernel-level threads) is acceptable for the traditional DES, the underlying operating system needs to be aware of the parallel threads in PDES [25]. Thus, only kernel-level threads (e.g., POSIX threads) are applicable in the synchronous PDES.

In the SystemC Language Reference Manual (LRM) [37], it clearly states that *"process instances execute without interruption"*. This requirement is also known as cooperative (or co-routine) multitasking in the SystemC execution semantics. As detailed in [26], this particular problem of parallel simulation is specifically addressed in the SystemC LRM:

> *"An implementation running on a machine that provides hardware support for concurrent processes may permit two or more processes to run concurrently, provided that the behavior appears identical to the co-routine semantics defined [...]. In other words, the implementation would be obliged to analyze any dependencies between processes and constrain their execution to match the co-routine semantics."*

We will describe the required dependency analysis for parallel SystemC simulation in Section 1.2, which is also needed for the Out-of-Order PDES.

1.1.3 Out-of-Order Parallel Discrete Event Simulation

In order to break the implicit barriers at the delta and timed cycle boundaries in the synchronous PDES, [12, 11] propose an Out-of-Order Parallel Discrete Event Simulation (OoO PDES) scheduling algorithm. Figure 1.4 depicts the OoO PDES scheduler for SystemC.

In OoO PDES, data structures and operations are refined as follows:

1. Each thread th is assigned a localized time stamp (t_{th}, δ_{th}).
2. Each event is assigned a notification time stamp (t_e, δ_e).
3. Time stamps are ordered in the following way:
    - $(t_1, \delta_1) = (t_2, \delta_2)$ iff $t_1 = t_2$, $\delta_1 = \delta_2$.
    - $(t_1, \delta_1) < (t_2, \delta_2)$ iff $t_1 < t_2$, or $t_1 = t_2$, $\delta_1 < \delta_2$.

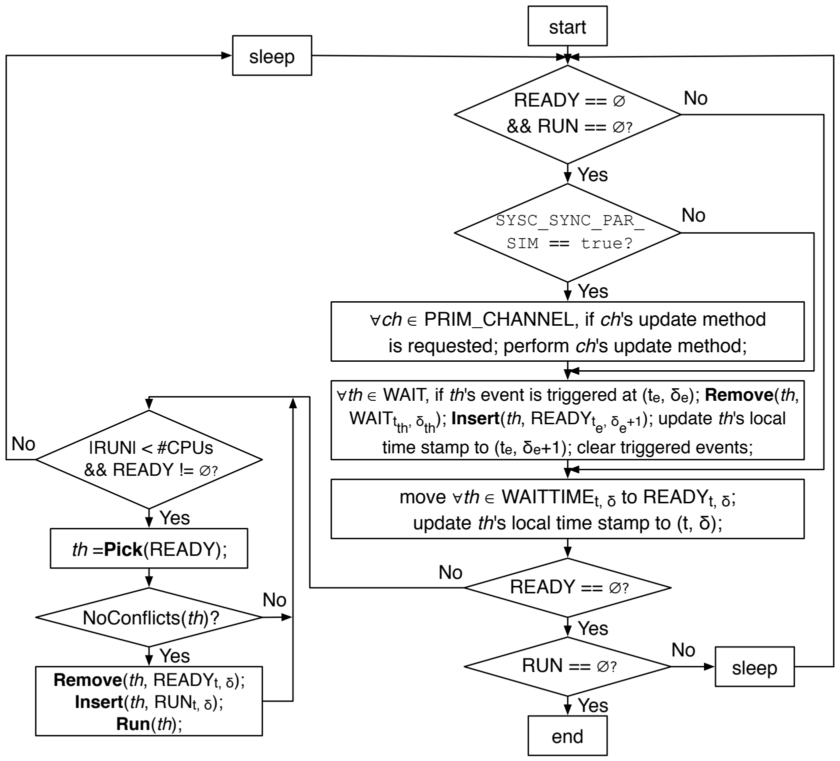

start
sleep
READY == ∅ && RUN == ∅?
No
Yes
SYSC_SYNC_PAR_SIM == true?
No
Yes
∀ch ∈ PRIM_CHANNEL, if ch's update method is requested; perform ch's update method;
∀th ∈ WAIT, if th's event is triggered at (te, δe); Remove(th, WAITtth, δth); Insert(th, READYte, δe+1); update th's local time stamp to (te, δe+1); clear triggered events;
move ∀th ∈ WAITTIMEt, δ to READYt, δ; update th's local time stamp to (t, δ);
No
READY == ∅?
Yes
RUN == ∅?
No
sleep
Yes
end
No
|RUN| < #CPUs && READY != ∅?
Yes
th =Pick(READY);
NoConflicts(th)?
No
Yes
Remove(th, READYt, δ); Insert(th, RUNt, δ); Run(th);

Figure 1.4: Out-of-Order Parallel Discrete Event Simulation (OoO PDES) scheduler for SystemC.

- $(t_1, \delta_1) > (t_2, \delta_2)$ iff $t_1 > t_2$, or $t_1 = t_2$, $\delta_1 > \delta_2$.

4. Thread queues are separated to multiple sets with different time stamps:

- QUEUES = {READY, RUN, WAIT, WAITTIME, COMPLETE}.
- READY = $\cup$READY$_{t,\delta}$, READY$_{t,\delta}$ = $\{th \mid th$ is ready to run at $(t, \delta)\}$.
- RUN = $\cup$RUN$_{t,\delta}$, RUN$_{t,\delta}$ = $\{th \mid th$ is running at $(t, \delta)\}$.
- WAIT = $\cup$WAIT$_{t,\delta}$, WAIT$_{t,\delta}$ = $\{th \mid th$ is waiting for an event (id_e, t_e, δ_e) since (t, δ), where $(t_e, \delta_e) \geq (t, \delta)\}$.
- WAITTIME = $\cup$WAITTIME$_{t,\delta}$, WAITIME$_{t,\delta}$ = $\{th \mid th$ is waiting for time advance to $(t, \delta)\}$.
- COMPLETE = $\cup$COMPELETE$_{t,\delta}$, COMPLETE$_{t,\delta}$ = $\{th \mid th$ completed its execution at $(t, \delta)\}$.

5. Initial state at the beginning of OoO PDES:

    - $\mathsf{THREADS} = \{th_{root}\}$, where $(t_{root}, \delta_{root}) = (0, 0)$.
    - $\mathsf{RUN} = \mathsf{RUN}_{0,0} = \{th_{root}\}$.
    - $\mathsf{READY} = \mathsf{READY}_{0,0} = \mathsf{WAIT} = \mathsf{WAIT}_{0,0} = \mathsf{WAITTIME} = \mathsf{WAITTIME}_{0,0} = \mathsf{COMPLETE} = \mathsf{COMPLETE}_{0,0} = \varnothing$.

In OoO PDES, every thread th has its own localized time stamp (t_{th}, δ_{th}), so that the global in-order event and simulation time updates are relaxed, allowing more threads (at different simulation cycles) to run in parallel and ahead of time [11, 56]. This results in a higher degree of parallelism and thus higher simulation speed.

Compared to the synchronous PDES in Figure 1.3, there is no more classic delta and timed cycles in Figure 1.4. Each thread performs the delta and timed cycles locally. Thus, due to the out-of-order scheduling and the eliminated central scheduling point for delta cycles, it is rather difficult to determine a safe point in OoO PDES scheduling when primitive channel update requests can be served. However, it is always possible to safely fall back to synchronous PDES (`SYSC_SYNC_PAR_SIM` equals `true` in Figure 1.4) when primitive channel updates are requested.

In Figure 1.4, the Out-of-Order PDES scheduling is aggressive. The scheduler moves threads from the WAIT queue to the READY queue whenever the READY and RUN queues become empty, and any threads in WAITTIME are moved to READY as soon as possible. Also, the scheduler dispatches one more thread for execution as long as an idle core and a ready thread without conflicts ($NoConflicts(th)$ is `true`) are available.

Algorithm 1 lists the pseudocode of conflict detection in the OoO PDES scheduler. Here, $NoConflicts(th)$ checks for any potential conflicts between Thread th and any other concurrent threads in RUN, READY and WAIT with an earlier or equal time. For each pair of

Algorithm 1 Conflict Detection in OoO PDES Scheduler

```
bool NoConflicts(Thread $th$) {
for all $th_2 \in \mathsf{RUN} \cup \mathsf{READY} \cup \mathsf{WAIT}$ where $(th_2.t,\, th_2.\delta) \le (th.t,\, th.\delta)$ do
    if Conflict($th$, $th_2$) then
        return false
    end if
end for
return true
}

bool Conflict(Thread $th$, Thread $th_2$) {
if $th$ has data conflict with $th_2$ then
    return true /* check data hazards */
end if
if $th_2$ may enter another segment before $th$ then
    return true /* check time hazards */
end if
if $th_2$ may wake up another thread to run before $th$ then
    return true /* check event hazards */
end if
return false
}
```

concurrent threads, *Conflict*(th, th_2) checks for any data, timing and event hazards between Threads th and th_2[1]. As we are using advanced static compile-time analysis and optionally dynamic run-time analysis (Section 1.2) to identify all the potential conflicts, these checks can be performed in constant time as simple table lookups.

1.2 Recoding Infrastructure for SystemC (RISC)

To realize the OoO PDES approach for the SystemC language, we propose the Recoding Infrastructure for SystemC (RISC) and describe the overall RISC Compiler and Simulator proof-of-concept prototype (Beta Release V0.3.0 as of 2016-09-30) in [56]. Currently, the

[1]Note that Algorithm 1 here is revised from Algorithm 2 in [11], in order to match the SystemC semantics.

RISC software is available as an open source package and can be downloaded freely from the following website [55]: `http://www.cecs.uci.edu/~doemer/risc.html`.

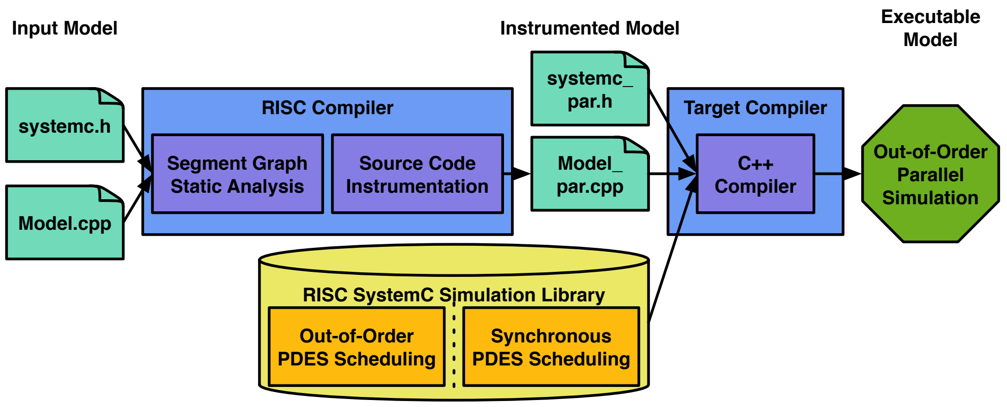

Input Model
systemc.h
Model.cpp
RISC Compiler
Segment Graph Static Analysis
Source Code Instrumentation
Instrumented Model
systemc_ par.h
Model_ par.cpp
Target Compiler
C++ Compiler
Executable Model
Out-of-Order Parallel Simulation
RISC SystemC Simulation Library
Out-of-Order PDES Scheduling
Synchronous PDES Scheduling

Figure 1.5: RISC Compiler and Simulator for OoO PDES of SystemC.

In contrast to the traditional SystemC simulation where a regular SystemC-agnostic C++ compiler includes the SystemC headers and links the input model directly against the SystemC library, we introduce a *dedicated SystemC compiler* to perform semantics-compliant SystemC simulation with maximum parallelism. As shown in Figure 1.5, our RISC compiler acts as a frontend that processes the input SystemC model and generates an intermediate model with special instrumentation for OoO PDES. The instrumented parallel model is then linked against our extended RISC SystemC library by the target compiler (a regular SystemC-agnostic C++ compiler) to produce the final executable model. OoO PDES is then performed simply by running the generated executable model.

From the user perspective, we essentially replace the regular SystemC-agnostic C++ compiler with the SystemC-aware RISC compiler (which in turn calls the underlying C++ compiler). Otherwise, the overall SystemC validation flow remains the same as before. It is just faster due to the parallel simulation.

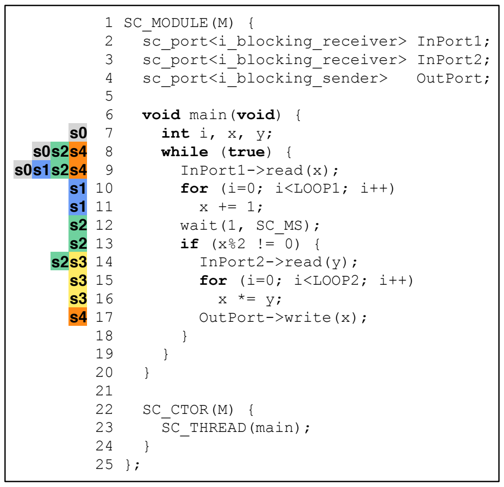
Internally, the RISC compiler performs three major tasks, namely Segment Graph construction, conflict analysis, and source code instrumentation. The simulator implements the semantics-compliant Out-of-Order PDES of SystemC, and falls back to synchronous PDES when the update methods in primitive channels are requested.

1.2.1 Segment Graph

The first task of the RISC compiler is to parse the SystemC input model into an Abstract Syntax Tree (AST) and then create a SystemC structural representation from the AST which reflects the SystemC module and channel hierarchy, connectivity, and other SystemC-specific relations, similar to the SystemC-clang representation [48, 69]. On top of this, the RISC compiler then builds a Segment Graph (SG) [12] data structure for the model.

In DES and PDES, threads switch back and forth between the states of *running* (threads in the READY and RUN queues) and *waiting* (threads in the WAIT and WAITTIME queues). A series of source code statements executed by a thread between two scheduling points can be defined as a thread *segment* [11]. Then, for a SystemC model, it can be converted to a corresponding *Segment Graph* (SG). The SG is a directed graph that represents the code segments executed during the simulation. The nodes in the SG are code segments and the edges indicate the transitions from one segment to another. The code segments always start from a SystemC scheduling primitive, e.g., `wait`, `SC_METHOD`, `SC_THREAD`, and `SC_CTHREAD`.

Figure 1.6 shows a SystemC thread with its segments and the corresponding SG. As shown, every segment starts with a scheduling primitive (including thread creation and context switches, e.g., the `SC_THREAD` in line 23 or the `wait` statement in line 12) and ends before another. The *read* and *write* functions in this example invoke a `wait` statement inside the function calls, so they are all blocking and start new segments (segment 1 and 3 in the *read* function and segment 4 in the *write* function). Also, as indicated in Figure 1.6a and 1.6b, one source code statement (e.g., the `while` statement in line 8) may belong to several

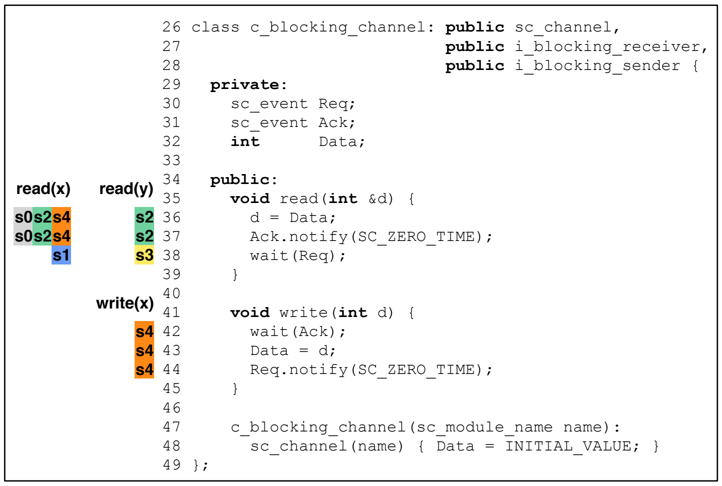


```
          SC_MODULE(M) {
            sc_port<i_blocking_receiver> InPort1;
            sc_port<i_blocking_receiver> InPort2;
            sc_port<i_blocking_sender>   OutPort;

            void main(void) {
      s0      int i, x, y;
  s0s2s4      while (true) {
s0s1s2s4        InPort1->read(x);
      s1        for (i=0; i<LOOP1; i++)
      s1          x += 1;
      s2        wait(1, SC_MS);
      s2        if (x%2 != 0) {
    s2s3          InPort2->read(y);
      s3          for (i=0; i<LOOP2; i++)
      s3            x *= y;
      s4          OutPort->write(x);
                }
              }
            }

            SC_CTOR(M) {
              SC_THREAD(main);
            }
          };
```

(a) Source code of an SC_THREAD.



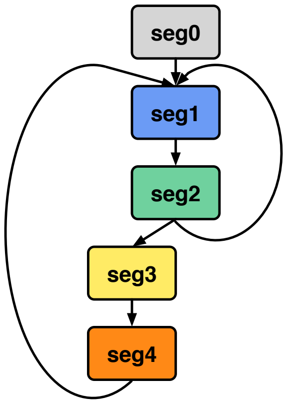
```
                     class c_blocking_channel: public sc_channel,
                                               public i_blocking_receiver,
                                               public i_blocking_sender {
                       private:
                         sc_event Req;
                         sc_event Ack;
                         int      Data;

read(x)   read(y)      public:
                         void read(int &d) {
s0s2s4        s2           d = Data;
s0s2s4        s2           Ack.notify(SC_ZERO_TIME);
    s1        s3           wait(Req);
                         }

         write(x)        void write(int d) {
              s4           wait(Ack);
              s4           Data = d;
              s4           Req.notify(SC_ZERO_TIME);
                         }

                         c_blocking_channel(sc_module_name name):
                           sc_channel(name) { Data = INITIAL_VALUE; }
                     };
```

(b) Source code of *read* and *write*.



(c) Segment graph.

Figure 1.6: SystemC thread and Segment Graph (SG) [57].

segments, depending on the execution paths. Here, the implementation of *c_blocking_channel* is simplified for demonstration purpose.

[11] presents a formal description of the Segment Graph and its construction algorithm, and [75] lists the detailed RISC Application Programming Interface (API).

1.2.2 Conflict Analysis

The segment graph data structure serves as the foundation for segment *conflict analysis*. As outlined in Algorithm 1, the OoO PDES scheduler must ensure that every ready thread to be dispatched has no conflicts (e.g., data hazards, timing hazards and event hazards) with any other threads currently in the READY, RUN and WAIT queues. Here, we utilize the RISC compiler to detect any potential conflicts between these threads with *static* analysis at compile time or *dynamic* analysis at run time.

- **Static Analysis**: Static analysis relies purely on the available information in the SystemC source code of the design model at hand. In this case, the RISC compiler performs very conservative identification of the potential hazards in the model, as outlined in [11]. Identifying all possible hazards is a complex analysis task that requires the full "understanding" of the module hierarchy. Here we statically extract the module hierarchy and analyze the individual threads.

- **Dynamic Analysis**: However, in most cases not all of the needed information can be gathered statically. For instance, design parameters may be passed via the command line to define the number of modules, certain channel characteristics, or other configuration information. In such SystemC models, the instantiated modules, channels, and ports are typically created through loops in a dynamic fashion. Thus, these exact parameters are only available at run time, so they cannot be statically analyzed. In these cases, dynamic analysis is needed.

  In dynamic analysis, the compilation flow is extended by a preprocessing step. The input SystemC model is fed into the RISC elaborator which produces an executable model that only performs the SystemC elaboration phase. At the end of the elaboration phase, the executable model automatically traverses the created module hierarchy via the SystemC introspection API and dumps this detailed structural design information into an instance connectivity file. This file is in turn provided as an input to the RISC compiler, so that the dynamically created design hierarchy and specific instance connectivity can be used for precise conflict analysis. The instance connectivity data file includes the actual module hierarchy, the specific port mapping, and the actual target variable mapping of references.

The dynamic analysis takes run-time information into account and augments the classic static analysis. The combination of static and dynamic analysis in the RISC compiler is called *hybrid* analysis [77].

1.2.3 Source Code Instrumentation

As a result of the conflict analysis (static, dynamic, or hybrid [77]), the RISC compiler generates several conflict tables that describe all possible conflicts between threads in any two segments. Using this conservative information, the simulator can then at run-time quickly determine by a simple table lookup whether or not it is safe to dispatch any given thread in parallel or ahead of time.

To pass information from the compiler to the simulator, we use automatic model instrumentation. That is, the intermediate model generated by the compiler contains instrumented (automatically generated) source code which the simulator can then rely on. At the same time, the RISC compiler also instruments user-defined SystemC channels with automatic protection against race conditions among communicating threads, as discussed in Section 1.1.2. Note that the source code instrumentation is performed automatically by the RISC compiler and no user-interaction is necessary.

In total, the RISC source code instrumentation includes four major components:

1. **Segment and instance IDs**: Individual threads are uniquely identified by a creator instance ID and their current code location (segment ID). Both IDs are passed into the simulator kernel as additional arguments to scheduling primitives, including `wait` and thread creation.

2. **Data and event conflict tables**: Segment concurrency hazards due to potential data conflicts and event conflicts are provided to the simulator as two-dimensional tables indexed by a pair of segment ID and instance ID.

3. **Current and next time advance tables**: Timing hazards between concurrent threads are passed to the RISC simulator as one-dimensional tables indexed by segment ID.

4. **User-defined channel protection**: SystemC allows users to define new channels for customized inter-thread communication. To ensure such communication is also safe in OoO PDES, the RISC compiler automatically protects user-defined channels (e.g., those derived from `sc_channel` and `sc_prim_channel`) by acquiring a channel lock (mutex) at the entry of the channel methods and releasing the lock at the exit. Thus, it is guaranteed that the execution of channel methods is mutually execlusive, and avoids the potential race conditions when communicating threads exchange data.

1.2.4 Compiler Backend

After the automatic source code instrumentation, the RISC compiler passes the generated intermediate model to the underlying regular C++ compiler. That target compiler then generates the final simulation executable by linking the instrumented code against the RISC extended SystemC library.

1.2.5 Simulator

Same as the Accellera proof-of-concept implementation of SystemC DES [86], the RISC simulator is not an explicit tool, but a run-time library [52] that the generated executable model is linked against. Thus, the Out-of-Order PDES is performed by executing the generated model from the target compiler.

By default, the RISC simulator follows the Out-of-Order PDES scheduling algorithm as outlined in Section 1.1.3. However, as soon as SystemC primitive channels are used with requests to the update methods, the simulator falls back to the synchronous PDES execution. Thus, as discussed in Section 1.1.3, such models will execute in safe synchronous mode.

As OoO PDES allows a higher number of threads running in parallel or even ahead of time, run-time scheduling optimization becomes critical for maximizing the simulation performance. Thus, we focus on the thread-to-core mapping and scheduling in this dissertation, and present details of various optimization algorithms in the following chapters.

1.3 Thread-to-Core Mapping

As discussed in Section 1.1.2 and 1.1.3, there are multiple threads running concurrently in the synchronous PDES and Out-of-Order PDES. A key challenge here is to determine the thread-to-core mapping for extracting maximum simulation performance [46, 17, 92]. Due to the shared resource contention on processing units and memory hierarchy, the simulation speed can be slowed down by more than 50% [92], and the scalable performance is not always readily available on multi-core and many-core hosts [18]. Also, the heterogeneity of the memory hierarchy may lead to varying communication latency, depending on whether the communication happens through local or remote caches, or even main memory [21]. Due to the complexity of the applications' behavior, the underlying resource topologies and the cache coherency protocols, determining the thread-to-core mapping that ensures the lowest contention and performance degradation [87, 81] has exponential complexity [19] and the problem of optimal thread-to-core mapping is NP-complete [44]. As an inefficient thread-to-core mapping also results in inefficent resource usage on multi-core and many-core hosts [45, 49], it is critical to efficiently optimize thread mapping in synchronous and Out-of-Order PDES.

In general, there are two categories of thread mapping methods, i.e., *static* and *dynamic*. The *static* methods usually leverage memory traces from binary instrumentation tools [71, 3, 23], which characterize the communication and computation loads for an application through profiling. Then, a static thread mapping is produced for the application to improve its run-time performance. Static mapping methods [9, 71, 3, 23] are simple to apply, and do not require to modify the source code of applications or support libraries. However, static thread mappings are not responsive to the run-time behavior of mapped workloads [54]. In comparison, *dynamic* thread mapping detects communication and tracks correlation between threads at run time, and performs online thread migration to cope with workload phase changes [21, 54]. It usually monitors page table accesses or page faults [22, 21, 88] to characterize the communication patterns, and then formulates the mapping problem as graph matching [22, 21] or 0-1 Integer Linear Programming (ILP) [46, 54]. In most cases, dynamic mapping performs better than static mapping at run time, but needs to be implemented at the OS kernel level [21, 22, 88].

In comparison to the general applications where any number of concurrent threads may arrive, execute and terminate in an unpredictable way [19], all the ready-to-run threads in SystemC PDES are available at the beginning of the delta cycle. Also, the behavioral and structural hierarchy of the ESL design model is clearly specified in SLDL (e.g., SystemC or SpecC) source code. Thus, the execution flow of PDES can be analyzed statically from the source code, and the same execution pattern repeats periodically[2]. In addition, both SLDL (SystemC and SpecC) implement their simulation libraries in the user level [86, 7] for

[2]In SystemC it is a common coding idiom to include an infinite loop within a `SC_THREAD` or a `SC_CTHREAD`, and a `SC_METHOD` executes its associate function from beginning to end whenever it is triggered [37]. SpecC has similar characteristics [29].

flexibility and portability, so dynamic mapping methods at the kernel level are not a good fit here. On the other hand, static methods using memory traces or binary analysis usually incur a high overhead [21]. Therefore, we propose a hybrid approach in this dissertation.

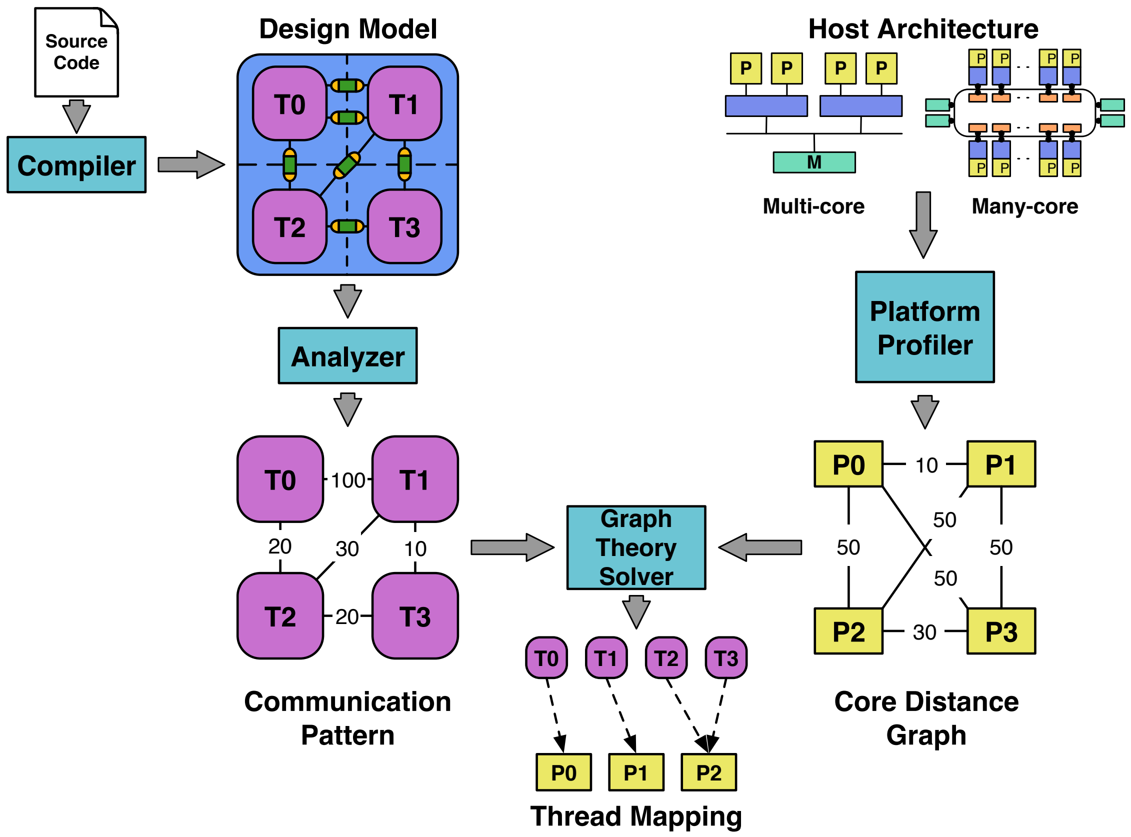


Figure 1.7: Problem decription of thread-to-core mapping in SystemC PDES.

Figure 1.7 depicts the problem of thread-to-core mapping in the context of SystemC PDES, together with our proposed approach. Here, we define our problem as follows:

With full source code of an ESL design model and no a priori knowledge of the underlying host architecture, optimize the thread-to-core mapping in SystemC Parallel Discrete Event Simulation. The goal here is to mitigate resource contention and communication latency and thus improve simulation performance.

In order to solve this problem, we propose to utilize our RISC compiler outlined in Section 1.2, and generate a communication pattern (presented in Section 4.4) to characterize the workload and communication in a design model. Then, as for the host architecture, we identify it with a concept of *core distance* (defined in Section 3.2) through profiling. Next, with two graphs (i.e., *Communication Pattern* and *Core Distance Graph* in Figure 1.7) available, a graph theory solver is used to generate an optimized thread mapping at run time. Compared to the previous thread mapping methods, our proposal is entirely implemented at the user level, and specifically designed for SystemC PDES. It guarantees sufficient accuracy and manageable overhead.

1.4 Related Work

Due to the inexpensive availability of parallel computing capabilities on multi-core and many-core hosts, Parallel Discrete Event Simulation (PDES) has become increasingly popular [27, 8, 67] during the past few decades. Despite all the PDES approaches follow the Discrete Event Simulation (DES) semantics and dispatch multiple threads concurrently, they differ in three major aspects: synchronization paradigms, abstraction levels, and host architectures.

Currently, there exist two major synchronization paradigms in PDES approaches, namely *conservative* and *optimistic*. Conservative PDES typically requires dependency analysis, and only dispatches threads that are safe to run concurrently. Either the conservative synchronous PDES scheduler [78] ensures in-order execution where the temporal barriers prevents effective parallelism, or the asynchronous approach [11] requires advanced compile-time conflict analysis (Section 1.2.2) to break the implicit barriers at the boundaries of delta and timed cycles. In contrast, optimistic PDES [43] assumes that threads are always safe to run and performs rollbacks if errors are detected.

Recalling Figure 1.1 which depicts the abstraction levels in system design, various PDES approaches also target different levels of abstraction. [34] compares different PDES approaches, including synchronous, asynchronous and cycle based simulation, at the Register Transfer Level (RTL). In order to speed up simulation, Transaction Level Modeling (TLM) [6] elevates the abstraction of communication and trade-offs timing accuracy against simulation speed. Also, source-level [84] and host-compiled [30] simulation abstract computation on the target platform to boost simulation speed. In addition, some approaches target mixed-abstraction levels (e.g., RTL and TLM in [79]).

Other PDES approaches are customized for specific host architectures. While most PDES approaches [78, 11, 63] target the common Symmetric Multiprocessing (SMP) machines, [80] emulates SystemC descriptions on FPGA and [66] partitions computation model in SystemC into concurrent threads on GPGPU. In [79], authors parallelize SystemC simulation across CPUs and GPUs.

As various PDES approaches allow multiple threads running concurrently on parallel processing units, it is critical to schedule these threads efficiently. In general, thread-to-core mapping is optimized for mitigating resource contention and communication latency in parallel execution. Two distinguished categories of mapping methods exist, namely static and dynamic. The static mapping methods [71, 3, 23] usualy use memory traces or binary analysis to characterize the computation and communication behaviors of an application, and generate a static thread mapping at compile time. In contrast, dynamic methods [54, 22, 21, 88, 46] profile applications at run time, and perform thread migration in response to the dynamic workload variation. [21] monitors accesses to page table and detects shared pages between threads, using this information as the communication pattern to dynamically migrate threads. In [19], authors propose an approach to first characterize applications offline, and then dynamically adjust the thread mapping at run time, based on the static application characteristics. [13] presents a run-time strategy that incorporates the user behavior information. In addition, thread mapping and data mapping are closely related. [22, 9, 46, 36] exploits thread mapping and data mapping simultaneously.

Different mapping methods are also customized for different host architectures. In [54] and [5], authors propose dynamic thread mapping strategies for heterogeneous multiprocessor systems. [51, 50] target General-Purpose Graphics Processing Units (GPGPU), and [88, 82] study Distributed Shared Memory (DSM) systems. Also, thread mapping for Networks-on-Chip (NoC) is a hot research topic [46, 13, 64, 9] in recent years. In [60, 15], authors take the characteristics of Non-Uniform Memory Access (NUMA) systems into consideration, and reduce the costly remote memory accesses via thread mapping. [35, 36] address the problem of thread mapping on Chip Multiprocessors (CMP).

In comparison to the previous mapping methods for general applications, we propose a PDES-specific approach in this dissertation. As outlined in Section 1.1, a number of threads are ready to run at the beginning of the delta cycle, and the behavioral and hierarchical structure of the design model is well presented in SLDL. Thus, we propose to first use our RISC compiler (Section 1.2) to analyze the communication patttern of the design model, and then optimize thread mapping at run time. Also, our approach requires no *a priori* knowledge of the underlying platform. It automatically profiles the host architecture and incorporates this information to the thread mapping optimization.

1.5 Goals and Overview

As outlined in Section 1.3, a good thread mapping for multiprocessor architectures (e.g., DSM, NUMA, and NoC) usually has the following requirements [88]:

1. **Load Balancing**: Load is uniformly distributed across different nodes (e.g., processors), and nodes' computational capacities match threads' computational needs.

2. **Communication Minimization**: The communication cost between threads located on distinct nodes are minimized.

Here, **Load Balancing** aims at mitigating the contention on shared resources, specifically the processing elements, on the host platform. In order to efficiently utilize the available computing units, none of them should be overloaded or underloaded. On the other hand, **Communication Minimization** targets the costly communication latency between separate nodes. To take advantage of the faster on-chip communication, it is necessary to map threads that share a high communication volume to processing cores on the same node. Both **Load Balancing** and **Communication Minimization** need to be taken into consideration in thread-to-core mapping.

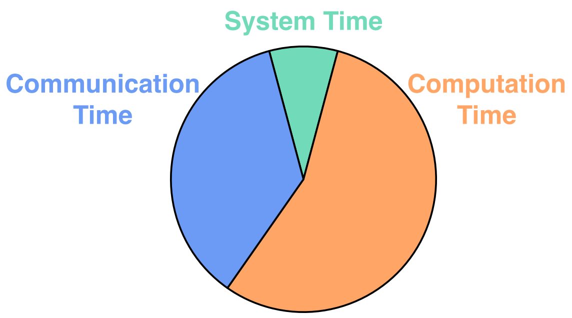


Figure 1.8: Decomposition of execution time of ESL simulation.

When running an ESL simulation on Linux, its execution time consists of two parts, namely system time and user time, when the simulation runs in kernel mode and user mode, respectively. In particular, user time can be further decomposed into computation time and communication time, according to the purpose of the execution. Figure 1.8 illustrates the typical decomposition of the execution time. Here, in order to optimize thread-to-core mapping for parallel ESL simulation, we would like to target different parts of the complete execution time.

First, the system time refers to the amount of time when the application executes in the kernel mode. It is mainly contributed by thread context switching and system I/O, and usually much shorter than user time. Note that it is difficult to accelerate the system time purely by user-level thread mapping. The most effective approach is to optimize the underlying thread library. Our previous work [53] proposes a hybrid thread library to reduce the overhead of thread creation/deletion and context switching.

Next, the computation time covers the amount of time on data crunching, mainly in the Arithmetic-Logic Unit (ALU) and Floating-Point Unit (FPU). When the total workload is determined by the specification and compiler, the total computation time varies significantly, depending on the dispatch order and thread-to-processor partitioning. **Load Balancing** improves computation time effectively.

Last, the communication time is composed of the communication latency between threads and the memory access. Notably, the memory hierarchy presents different access speeds on different levels, and the on-chip communication is always faster than the inter-chip one. Thus, **Communication Minimization** is a promising approach to shorten communication time.

Section 1.3 defines our problem of thread mapping. More specifically, our goals include:

1. **Reducing the Computation Time in SystemC PDES**: For those computation-intensive applications and examples, we would like to optimize load balancing in PDES through thread mapping, so as to reduce the computation time effectively.

2. **Mitigating the Communication Time in SystemC PDES**: For those communication-intensive examples, it is critical to mitigate the costly on-chip and inter-chip communication, and map threads that share a high amount of communication to cores on the same chip.

3. **Decreasing the Total User Time in SystemC PDES**: Finally, we target the general-purpose application, in which both computation and communication are significant. Here, we try to minimize the complete user time in SystemC PDES, by taking both **Load Balancing** and **Communication Minimization** into consideration.

In the following chapters, we present our approach to realize these goals. Specifically, Goal 1 is addressed in Chapter 2, and Goal 2 is achieved in Chapter 3 and 4. In Chapter 5, we propose an integrated algorithm to meet Goal 3.

The rest of the dissertation is organized as follows:

In Chapter 2, we propose a dynamic load-profiling and *segment-aware* scheduling algorithm with optimized thread dispatching to balance workloads on parallel processing units. Based on a compile-time generated Segment Graph (SG), our scheduler [57] can accurately predict the run time of the thread segments ahread and thus make better dispatching decisions. In the evaluation, our segment-aware scheduler consistently shows a significant performance gain compared to the previous scheduling policies.

In Chapter 3, we first define the concept of *core distance* for multi-core and many-core architectures, and observe that for many-core architectures using directory-based cache coherence protocols, the core-to-core communication latency depends not only on the physical placement on the chip, but also on the location of the distributed cache tag directory. Using a ping-pong memory benchmark, we quantify the core distance of a ring-network platform, and propose an algorithm [58] to optimize the thread-to-core mapping in order to minimize on-chip communication overhead. In our experiments, our algorithm speeds up communication-intensive benchmarks by more than 25% on average over the Linux default mapping strategy.

Based on the core distance concept from Chapter 3, we explore thread mapping for the general multiprocessor architectures in Chapter 4. By analyzing the communication pattern and profiling the processor architecture, our proposed framework formulates the problem of thread mapping as graph partitioning and automatically generates an optimized thread-to-core mapping for the target architecture. Our framework is *not* customized for a specific architecture, and reduces the costly inter-chip communication. In the comprehensive evaluation, our approach shows a performance gain of up to 28% with negligible overhead.

In Chapter 5, we integrate our thread mapping techniques in Chapter 2 and 4 to reduce the total user time for general design models, in which both computation and communication are intensive. The experimental results show that our integration performs better than both techniques from Chapter 2 and 4 for a real-world application, and achieves an additional speedup of 10% compared to the previous two methods.

Finally, Chapter 6 concludes this dissertation with a summary of our contributions and the future work.

Chapter 2

Computation-Aware Thread Mapping

In this chapter, we first propose a dynamic load-profiling and *segment-aware* scheduling algorithm with optimized thread dispatching to reduce the computation time in SystemC PDES. Based on a static segment graph from our RISC compiler (Section 1.2), our segment-aware scheduler [57] accurately predicts the run time of the thread segments ahead, and makes better dispatching decisions for load balancing.

2.1 Introduction

Discrete Event Simulation (DES) has been in use for decades to validate the functionality of Electronic System Level (ESL) designs. In order to improve the performance of DES, Parallel Discrete Event Simulation (PDES) [27] was proposed to run threads in parallel. With the popularity of multi-core hosts, parallel computing platforms are readily available and provide great potential to achieve better performance.

Currently, the SystemC [37] System-Level Description Language (SLDL) is used for system design as an IEEE standard. However, the reference simulation library of SystemC still relies on DES, running a single thread at any time, and cannot utilize the computing capabilities of parallel platforms. In recent years, a lot of parallel SystemC simulation approaches [78, 34, 79, 63] have been proposed, which speed up simulation significantly due to parallel execution. However, very few of these approaches [14] address the load balancing problem in their parallel schedulers. In this chapter, we propose a segment-aware multi-core thread dispatch algorithm [57], which can be applied to all work-sharing PDES (SystemC, SpecC, etc.) schedulers. By parsing a design model to a graph of thread segments (a portion of source-code statements between two scheduling points) using static compiler analysis and profiling segment execution at runtime, our approach automatically optimizes the thread dispatch order and consistently achieves a significant speedup over previous thread dispatchers.

The key contributions of this chapter are the following:

1. We identify Longest Job First (LJF) as a better than default thread dispatch policy, when thread run time prediction is available.
2. We propose a novel technique to accurately predict thread run times based on a static Segment Graph (SG) and the specific segments threads will execute.
3. We evaluate our proposed segment-aware approach in comprehensive experiments and show that it consistently improves performance for both synthetic and real-world examples.

The rest of the chapter is organized as follows: Section 2.2 reviews background on parallel SystemC simulation and related work on load-balancing optimizations in PDES. In Section 2.3, we introduce our parallel SystemC implementation, then discuss multi-core scheduling, and propose our optimization algorithm in Section 2.4. In Section 2.5, we evaluate our segment-aware algorithm with both synthetic and real-world examples. Section 2.6 concludes the chapter.

2.2 Background and Related Work

Parallel SystemC simulation has been a hot research topic in the past few years. In general, these parallel approaches differ in the simulation strategies they apply, the abstraction levels they target, and the host architectures they use. [78] proposes a conservative synchronous parallel simulation approach in which a master thread performs the update and notification phases and a pool of worker threads execute parallel SystemC processes. [34] compares different parallel SystemC approaches, e.g., synchronous PDES, asynchronous PDES, and cycle based simulation, at the Register Transfer Level (RTL). Both [78] and [34] target shared-memory multi-core host architectures. In order to further boost the simulation speed, [79] partitions mixed-abstraction RTL and Transaction Level Models (TLM) into processes suitable for GPU and CPU execution. In [63], the author proposes an approach that explicitly targets loosely timed systems, and runs parallel processes at different simulation cycles. In comparison, [11] proposes a conservative asynchronous PDES approach that also perserves cycle accuracy. The proposed approaches in [63] and [11] run on multi-core hosts.

In this chapter, we implement a synchronous PDES approach similar to [78], and propose a segment-aware thread dispatcher inside the PDES scheduler. Our proposed dispatcher is orthogonal to the above approaches, and can be applied to any work-sharing PDES schedulers for shared-memory multi-core machines.

Compared with the many parallel SystemC simulation approaches, load-balancing optimizations on thread dispatching in the context of PDES have gained little attention. [73] presents a dynamic load migration algorithm for reducing the total number of rollbacks in an optimistic PDES environment. [14] proposes a novel parallel SystemC simulation approach with hierarchical multithreading, and optimizes load balancing by using workload stealing. In [91], authors improve dynamic load balancing of PDES by using the proposed Random, Communication-based and Load-based (RCL) load migration policies. [38] evaluates different process partition strategies (user defined, hash-based, round-robin, etc.) to improve load balancing of parallel simulation. However, in the previous work, the model is divided into distributed logical OS processes and they are allocated to different processors. To avoid workload imbalance on different processors, all the previous work focuses on algorithms to transfer workload among different processes, which is different from ours for work-sharing simulators. To the best of our knowledge, this chapter proposes the first scheduler for work-sharing SystemC PDES with thread dispatch order optimized based on the static analysis of the model at hand.

2.3 Parallel SystemC Simulation

The SystemC reference simulator is based on DES. As outlined in Section 1.1.1, a single thread is running at all times in the traditional DES scheduler. When all runnable threads in the READY queue finish their current delta cycle, the root thread resumes and performs the update and notification phases. Then the simulation proceeds to the next delta cycle. If no more threads are runnable after the update and notification, the current time cycle finishes. The simulator advances the time and processes the earliest timed event from the WAITTIME queue. When the READY and WAITTIME queues are both empty, the simulation ends.

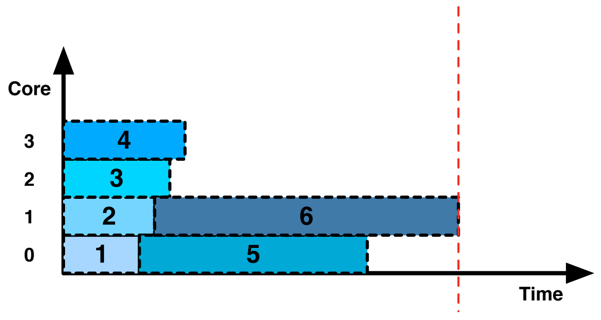
In contrast to DES, a PDES scheduler (Section 1.1.2) dispatches multiple runnable threads concurrently onto multiple available processor cores. In the evaluation phase of our parallel implementation, one more thread (`SC_METHOD`, `SC_THREAD` or `SC_CTHREAD`) is dispatched from the READY queue and starts its execution, as long as an idle processor core is available. When all threads finish their evaluation phase, the last running thread performs the update and notification phases and advances the time if the READY queue is still empty.

In order to avoid race conditions among accesses to internal shared states, we protect the SystemC kernel with a mutex lock, as discussed in Section 1.1.2. Whenever a thread needs to modify the state of the kernel, it first acquires the mutex before entering the critical section. Similar to [78] and the synchronous approach in [34], our parallel SystemC implementation here only executes threads from the same delta cycle in parallel and has a central READY queue. For safe communication, our approach also automatically protects every channel with a local lock [26] and fully supports the standard SystemC semantics, including immediate notification. Differing from [78], our parallel kernel is symmetric and every thread can perform the scheduling functions. Thus, our implementation does not need a special root thread to perform the update and notification phases, which eliminates the frequent context switches from worker threads to the root thread.

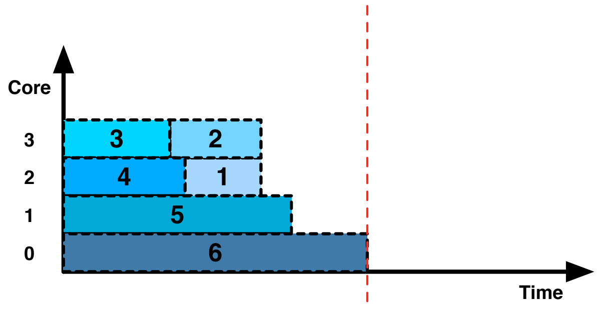
Also, a semantics-compliant SystemC implementation is *"obliged to analyze any dependencies between processes and constrain their execution to match the co-routine semantics"* [37]. Here, we rely on the dedicated RISC compiler (Section 1.2) to identify potential data hazards inside design models in order to avoid any race conditions on shared variables. As outlined in Section 1.2, our SystemC compiler [56] performs three major tasks, namely conflict analysis, segment-graph construction, and source-code instrumentation. For this chapter, we extend and exploit the segment-graph analysis and the corresponding code instrumentation to accurately predict the next thread run times, so that the dispatcher in the scheduler can quickly make better decisions. We present more details of our SystemC compiler in Section 1.2, which are also recapped in Section 2.4.3.

2.4 Multi-Core Scheduling

In each delta cycle of PDES, a number of threads are available in the READY queue, as determined by the PDES scheduler. However, these threads typically have a diverse run time in the evaluation phase. We note that the order of dispatching these threads on a multi-core host has a significant influence on the total execution time. As an illustrative example, Figure 2.1 compares the execution time of two classic dispatch policies, namely Shortest Job First (SJF) and Longest Job First (LJF), on a four-core machine. In the order determined by their (predicted) run time, threads are assigned to the available CPU cores. When employing LJF (Figure 2.1b), the current evaluation phase finishes much earlier than when using SJF (Figure 2.1a).



(a) Shortest Job First (SJF).



(b) Longest Job First (LJF).

Figure 2.1: Two multi-core thread dispatch policies.

In general, multi-core scheduling is a classic load balancing problem in algorithm design, which decides the thread dispatch order and is orthogonal to the parallel DES approach. Following this, we propose a *segment-aware* LJF-based thread dispatch optimization to improve the performance of parallel SystemC simulation for the case that the number of parallel threads in the model is greater than the number of cores on the host. Note that our key contribution here is not LJF itself, but the accurate prediction of thread run times that LJF depends on.

2.4.1 Classic LJF Thread Dispatch Policy

Without loss of generality, we will assume Linux as the underlying OS on which the parallel SystemC simulator is based. For Linux, the default scheduling policy since version 2.6.23 is Completely Fair Scheduling (CFS). Even though CFS maintains fairness of execution time among threads, it always picks first the thread which previously took the least processor time. Thus, it favors interactive processes over batch processes (e.g., PDES simulation). CFS is similar to the SJF policy, except in Linux each core has a separate READY queue. If Linux detects an unbalanced load on different cores, thread migration is used for balancing.

The general multi-core scheduling problem is proven to be NP-complete in the literature [68, 74]. Thus, in order to efficiently generate an optimized thread dispatch policy for parallel SystemC simulation, the well-established LJF policy should perform better than the default Linux dispatch policy, as illustrated in Figure 2.1. In each evaluation phase of the parallel simulation, our kernel measures the run time of each thread by reading the CPU cycle count registers, which has minimal overhead, and uses this profiling information as the run time prediction for the next evaluation phase[1]. Then, the dispatcher sorts threads in the READY queue in decreasing order based on their previous execution time. Thus, the threads estimated to run the longest will run first on the available cores. Shorter threads are dispatched then whenever a core becomes available. With m denoting the number of available processor cores, this greedy scheduling algorithm has been proven to introduce a slowdown of less than $(\frac{4}{3} - \frac{1}{3m})$ compared with the optimal multi-core scheduling [32].

The classic LJF thread dispatch policy works well when threads show identical run time every time they are issued. However, this typically is *not* the case in SystemC simulation.

[1]Even though the input data to the operations in a thread may vary in different runs, the thread run time likely stays similar. Also, the LJF dispatcher only needs to know the relative order of any two threads' workload, e.g. thread 1 runs longer than thread 2, rather than the absolute values. Therefore, we estimate the next execution time of a thread to be the same as the previous one. Note that it is possible to apply other methods to predict thread run times, but this is not the focus of this chapter.

A SystemC thread performs an overall job, but such job usually consists of very different tasks. Thus, the actual run time of a thread in the simulation depends on its specific next task ahead (e.g. reading input data, processing frames, or sending output data). Taking this observation into account, we distinguish between the *segments* of code that a given thread executes, and propose our segment-aware dispatch algorithm.

2.4.2 Segment Graph (SG)

As discussed in Section 1.2.1, threads switch back and forth between the states of *running* (threads in the READY and RUN queues) and *waiting* (threads in the WAIT and WAITTIME queues) in PDES. A series of source code statements executed by a thread between two scheduling points can be defined as a thread *segment* [11]. Then, for a SystemC model, it can be converted to a corresponding *Segment Graph* (SG). The SG is a directed graph that represents the code segments executed during the simulation. The nodes in the SG are code segments and the edges indicate the transitions from one segment to another. The code segments always start from a SystemC scheduling primitive, e.g., `wait`, `SC_METHOD`, `SC_THREAD`, `SC_CTHREAD`, etc.. Thus, our RISC compiler can instrument these scheduling primitives with segment IDs in the source code. Then, when a thread resumes execution from the *waiting* state, the dispatcher can identify the current segment (before actually running it).

In general, every thread is composed of one or multiple segments. The adjacent segments perform different functions (e.g., in Figure 1.6a reading input data in segment 0 and processing it in segment 1) and usually run for a different amount of execution time.

2.4.3 Dedicated SystemC Compiler

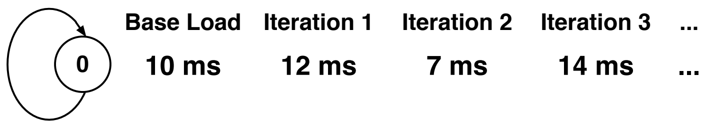
In order to identify the segment structure of a SystemC model and analyze the interdependencies between threads, we adopted the RISC compiler for our parallel SystemC simulation framework. The RISC compiler first parses an input SystemC model into ROSE [70] Abstract Syntax Tree (AST) and constructs a Segment Graph (SG) on top of the AST, following the Algorithm 3 in [11]. Here, we treat the `SC_METHOD`, `SC_THREAD` and `SC_CTHREAD` in the SystemC model as thread creation points (*STMNT_PAR* in Algorithm 3 [11]). Then, according to the SG, the compiler will append the segment ID as an extra argument to the AST nodes of the segment boundary primitives (e.g., `wait`, `SC_METHOD`, `SC_THREAD`, and `SC_CTHREAD`). For those function calls that start new segments (e.g., the *read* function in Figure 1.6a), our compiler modifies the function prototype of the called function, adding an integer variable of the segment ID as a new argument. Next, in the function definition, the segment boundary primitives will use the new argument to identify the next segment. Also, based on the SG, the compiler automatically performs static conflict analysis between segments and passes conflict tables to the parallel simulator. In addition, to ensure safe communication, the compiler protects user defined channels (e.g., those derived from `sc_channel` and `sc_prim_channel`) by acquiring a local channel lock at the entry of their public functions and releasing the lock at the exit. The details of static conflict analysis and channel protection can be found in Section 1.2.

2.4.4 Segment-Aware Dispatch Algorithm

Figure 2.2 depicts three typical segment structures inside a thread. In Figure 2.2a, the thread contains a single segment in a loop[2]. For different iterations, the workload of the segment is similar and typically varies only little due to the input data. Here, the base load in the figure

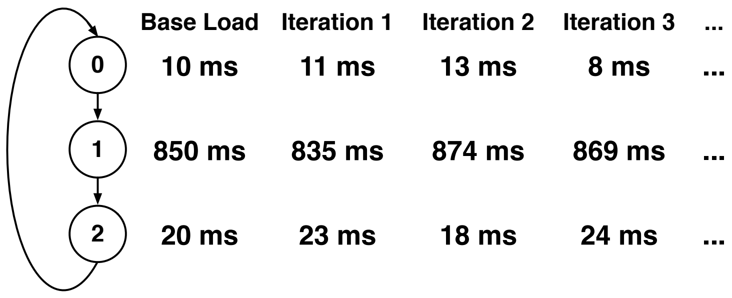
[2]The loop structure is default for `SC_METHOD` and a common coding idiom for `SC_THREAD` and `SC_CTHREAD` [37].

denotes the average amount of execution time of the segment. Figure 2.2b shows a thread that is composed of three consecutive segments in a loop, e.g., input, processing, and output. The three segments have different workload and their execution time varies significantly. Compared with the cyclic Figure 2.2a and 2.2b, the segment structure in Figure 2.2c is more general. Threads may have different execution paths in different situations. While the transition between their segments may be unpredictable, we find that a given segment typically carries a similar workload every time it runs.



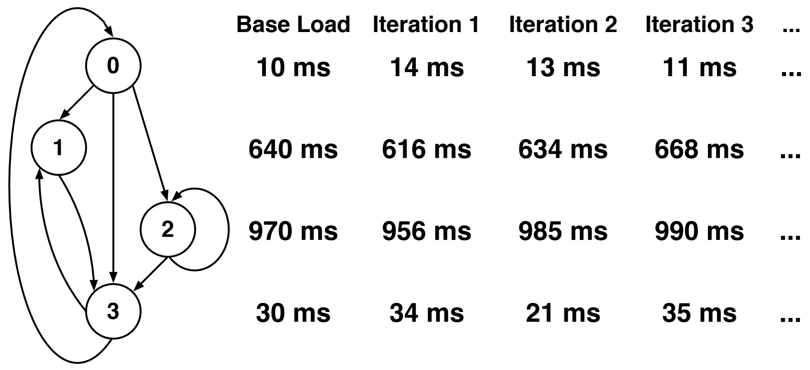
Base Load	Iteration 1	Iteration 2	Iteration 3	...
10 ms	12 ms	7 ms	14 ms	...

(a) A single segment in a cyclic thread.



Base Load	Iteration 1	Iteration 2	Iteration 3	...
10 ms	11 ms	13 ms	8 ms	...
850 ms	835 ms	874 ms	869 ms	...
20 ms	23 ms	18 ms	24 ms	...

(b) Consecutive segments in a cycle.

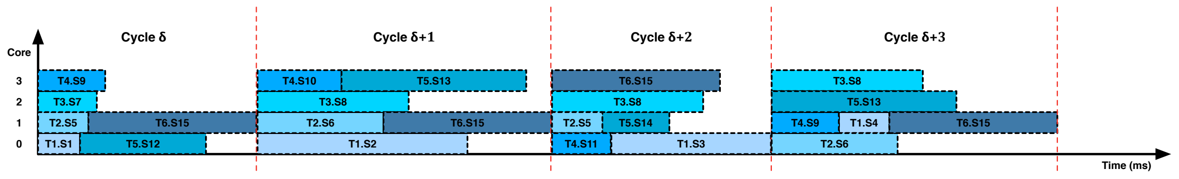


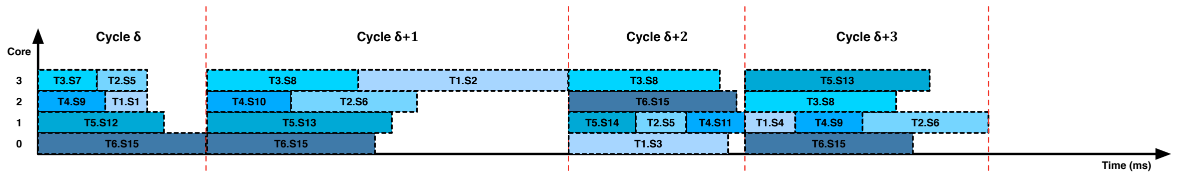
Base Load	Iteration 1	Iteration 2	Iteration 3	...
10 ms	14 ms	13 ms	11 ms	...
640 ms	616 ms	634 ms	668 ms	...
970 ms	956 ms	985 ms	990 ms	...
30 ms	34 ms	21 ms	35 ms	...

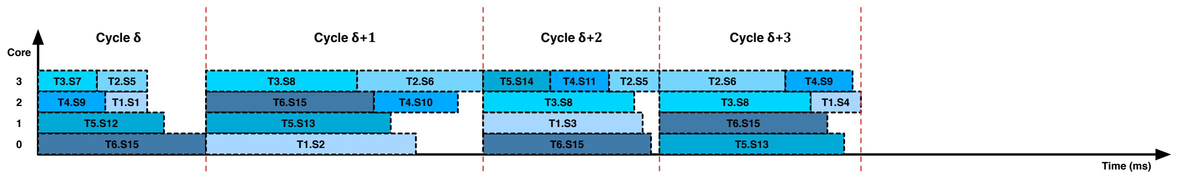
(c) Multiple segments in a general thread.

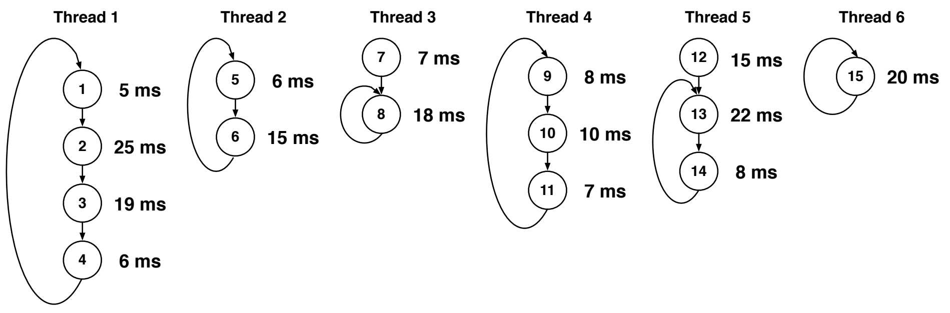
Figure 2.2: Different segment structures in a thread.

The classic LJF thread dispatch policy with load prediction based on the previous run only performs well for the simple segment structure in Figure 2.2a. However, for segment structures 2.2b and 2.2c, LJF will rely on wrong predictions and thus perform badly. For


(a) Shortest Job First (SJF) thread dispatch order.


(b) Longest Job First (LJF) thread dispatch order.


(c) Segment-aware thread dispatch order.

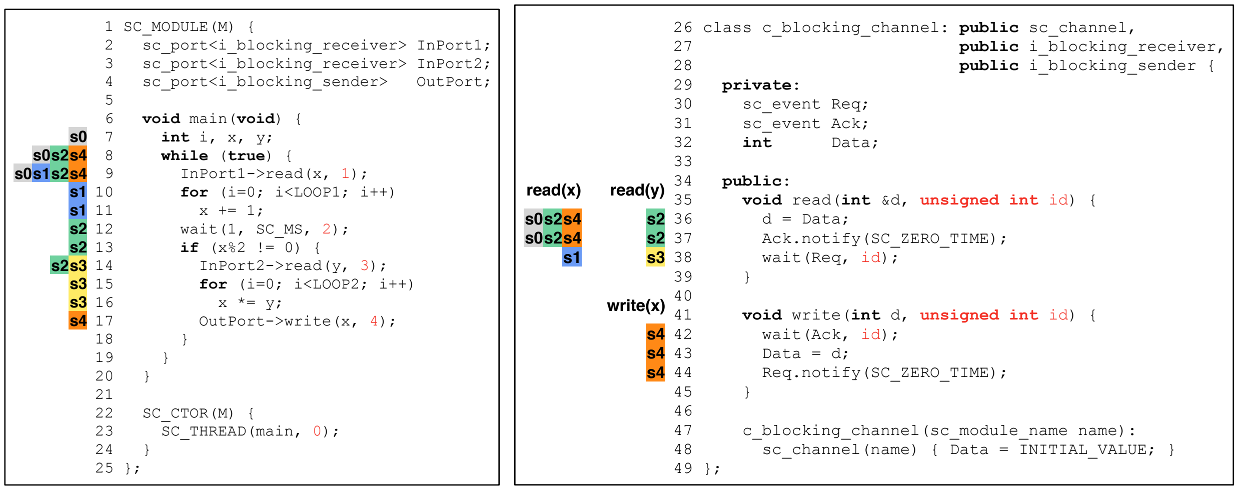


(d) Segment graphs of the six threads.

Figure 2.3: Thread execution timelines determined by different thread dispatch policies.

example, if a thread has the segment structure of Figure 2.2b, LJF will use the run time of segment 0 to predict the execution time of segment 1. Since the classic LJF policy is unaware of the segment structure, it treats the segments the same. However, the workload of the segments is unrelated and varies. Thus, the predicted run time for the next thread segment is inaccurate and LJF performs poorly.

Figure 2.3a, 2.3b and 2.3c compare the thread execution timelines determined by different dispatch policies. Here $Ti.Sj$ denotes Segment j in Thread i. Figure 2.3d depicts the segment structures in all threads. The numbers next to the nodes in Figure 2.3d are the workloads of the segments. Here, we assume that the prediction in Cycle δ is precise in all three cases, and the workload of a segment stays the same every time it runs. As shown, whereas SJF performs poorly in all four cycles, LJF only performs well in Cycle δ and $\delta+2$. The performance degrades in Cycle $\delta+1$ and $\delta+3$ due to the wrong prediction based on previous segments.



```
               1 SC_MODULE(M) {
               2   sc_port<i_blocking_receiver> InPort1;
               3   sc_port<i_blocking_receiver> InPort2;
               4   sc_port<i_blocking_sender>   OutPort;
               5
               6   void main(void) {
          s0   7     int i, x, y;
      s0s2s4   8     while (true) {
    s0s1s2s4   9       InPort1->read(x, 1);
          s1  10       for (i=0; i<LOOP1; i++)
          s1  11         x += 1;
          s2  12       wait(1, SC_MS, 2);
          s2  13       if (x%2 != 0) {
        s2s3  14         InPort2->read(y, 3);
          s3  15         for (i=0; i<LOOP2; i++)
          s3  16           x *= y;
          s4  17         OutPort->write(x, 4);
              18       }
              19     }
              20   }
              21
              22   SC_CTOR(M) {
              23     SC_THREAD(main, 0);
              24   }
              25 };
```

```
                     26 class c_blocking_channel: public sc_channel,
                     27                           public i_blocking_receiver,
                     28                           public i_blocking_sender {
                     29   private:
                     30     sc_event Req;
                     31     sc_event Ack;
                     32     int      Data;
                     33
read(x)   read(y)    34   public:
                     35     void read(int &d, unsigned int id) {
s0s2s4          s2   36       d = Data;
s0s2s4          s2   37       Ack.notify(SC_ZERO_TIME);
    s1          s3   38       wait(Req, id);
                     39     }
                     40
          write(x)   41     void write(int d, unsigned int id) {
                s4   42       wait(Ack, id);
                s4   43       Data = d;
                s4   44       Req.notify(SC_ZERO_TIME);
                     45     }
                     46
                     47     c_blocking_channel(sc_module_name name):
                     48       sc_channel(name) { Data = INITIAL_VALUE; }
                     49 };
```

(a) Source-code instrumentation of an SC_THREAD.

(b) Source-code instrumentation of *read* and *write* functions.

Figure 2.4: SystemC source-code instrumentation.

In order to accurately follow the segment structure, we utilize our dedicated SystemC compiler (Section 1.2) to generate the SG of the model and instrument the segment boundary primitives with a segment ID as an extra argument. Figure 2.4 shows the source code instrumentation of the SystemC thread in Figure 1.6. Here, for example, line 12 in Figure 1.6a is transformed to `wait(1, SC_MS, 2)`, where 2 is the segment ID. Then the scheduler is aware of the current segment of the runnable threads and can accurately predict their execution time based on the profiling information for the given segment.

Algorithm 2 lists the pseudocode of our segment-aware scheduling algorithm. The calling thread first reads the CPU cycle count register and records the run time of the current segment. Then, the segment ID of the current thread is updated to the next one. Next, while any threads exist in the READY queue, our scheduler will dispatch them in order to any available cores, and resume their thread execution. If no core is available but the READY queue has remaining threads, the current thread suspends itself. As the sequential scheduler, when the READY queue becomes empty, our scheduler performs the update and notification to start a new delta or timed cycle. Before the beginning of the new cycle, the dispatcher sorts threads in the READY queue in descending order of their previous run time in the same segment. For the very first evaluation phase of a segment, the algorithm can use either static compiler analysis, user input, or random values as prediction. Then, in later evaluation phases, the dispatcher predicts the thread execution time using the profiling time of the same segment, instead of the previous run time of the same thread in the classic LJF dispatcher.

Figure 2.5 depicts the software hierarchy of our parallel SystemC simulation framework with the segment-aware scheduler and dispatcher. Note that the thread dispatcher is implemented inside the PDES scheduler of the SystemC simulation library (user level), and we do not modify the kernel-level OS scheduler. Compared with the case that the regular parallel SystemC simulator dispatches *all* runnable threads and lets the Linux OS scheduler determine the thread execution, our segment-aware dispatcher only dispatches a number of threads equal to the number of available cores, and fixes their core affinity. Thus, our dispatcher is in full control and the Linux OS scheduler will not modify the thread execution order in our parallel simulation.

Using the segment-aware prediction, our scheduling algorithm can generate an optimized thread dispatch order[3]. Figure 2.3c shows that this is much better than treating all segments the same. Since the prediction is based on the correct segment (and we assume that the dispatcher already collects the profiling information of all segments), our segment-aware dispatcher achieves the best of the three thread dispatch policies in all four displayed cycles.

[3]In the case of a single segment per thread, as in Figure 2.2a, the segment-aware approach falls back to the classic LJF.

Algorithm 2 Segment-Aware Scheduling Algorithm

```
Input:
    Current thread $th_{curr}$
    Next segment $SegID_{next}$
 1: $th_{curr}.T_{end} \leftarrow$ CurrentCycles()
 2: $RunTime[th_{curr}.SegID] \leftarrow th_{curr}.T_{end} - th_{curr}.T_{start}$
 3: $th_{curr}.SegID \leftarrow SegID_{next}$
 4: while true do
 5:   while READY $\neq \varnothing$ do
 6:     if $\exists c \in$ Cores where $c$ is idle then
 7:       $th_{next} \leftarrow$ pop(READY)
 8:       $th_{next}.T_{start} \leftarrow$ CurrentCycles()
 9:       dispatch($th_{next}$, $c$)
10:     else
11:       suspend($th_{curr}$)
12:     end if
13:   end while
14:   Delta cycle $\delta \leftarrow \delta + 1$
15:   process any requested updates in primitive channels
16:   process any delta notifications
17:   if READY $= \varnothing$ then
18:     advance simulation time
19:     process any timed notifications
20:     if READY $= \varnothing$ then
21:       terminate the simulation
22:     end if
23:   end if
24:   sort threads $\forall th \in$ READY in decreasing order of $RunTime[th.SegID]$
25: end while
```

Since the prediction is based on the correct segment, our segment-aware dispatcher will achieve better performance than the classic LJF.

Note that the starvation problem for short threads cannot happen in our segment-aware scheduling algorithm for PDES, since no other threads will be added to the READY queue in between a delta cycle, unless an immediate notification occurs. However in the Accellera sequential simulation library and our parallel implementation, even the immediate notified threads will be made runnable only after all current threads in the READY queue are dispatched.

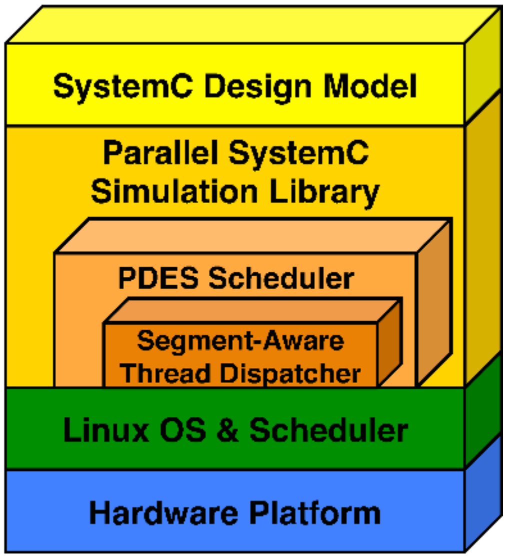


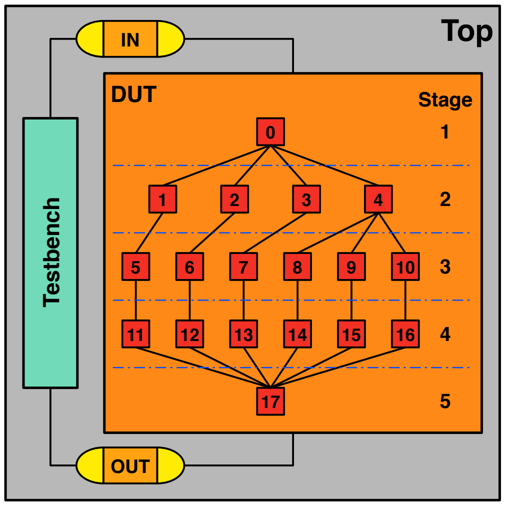
Figure 2.5: Software hierarchy of our parallel SystemC simulator implementation.

2.5 Experimental Evaluation

We now evaluate our segment-aware optimization on parallel SystemC simulation with synthetic benchmarks and real-world examples to demonstrate the performance gain. Table 2.1 lists the hardware specifications of the two multi-core workstations we used in our experiments.

Table 2.1: Workstations used for experiments.

Host	**8-Core**	**32-Core**
Processor	Intel Xeon E3-1240	Intel Xeon E5-2680
CPU frequency	3.4 GHz	2.7 GHz
Physical CPUs	1	2
Cores/CPU	4	8
Hyperthreads/core	2	2
Total HW threads	8	32



(a) Task graph block diagram.

```
1  SC_MODULE(TASK0) {
2    sc_fifo_in<FLOAT_T> iport0;
3    ...;
4    sc_fifo_out<FLOAT_T> oport0;
5    ...;
6
7    void main(void) {
8      while (true) {
9        FLOAT_T x0 = iport0->read();
10       FLOAT_T y0 = INITIALVAL;
11       ...;
12
13       for (i=0; i<ITER0; i++)
14         y0 = y0 * y0 + y0;
15       ...;
16       oport0->write(y0 + x0);
17       ...;
18     }
19   }
20
21   SC_CTOR(TASK0) {
22     SC_THREAD(main);
23   }
24 };
```

(b) Source code of a task.

Figure 2.6: Synthetic SystemC benchmark models.

2.5.1 Task Graph Benchmarks

Task Graphs For Free (TGFF) [20] is a popular tool to generate standardized random benchmarks for scheduling and allocation research. For our evaluation, we extended TGFF to output actual SystemC models of the generated task graphs. In particular, every task in the task graph is converted to an `SC_MODULE` which initiates an `SC_THREAD` with a specific amount of workload. Here, `SC_THREAD` is used as a test case, but our approach is applicable to `SC_METHOD` and `SC_CTHREAD` as well. Between the tasks, the `SC_MODULE`s use `sc_fifo` channels to communicate and synchronize. Figure 2.6 shows the block diagram of a generated SystemC model and the code of a task (a block in Figure 2.6a). Next, we evaluate the three categories of thread segment structures introduced in Figure 2.2 separately.

2.5.1.1 Single-Segment Threads

For the first experiment, we use the extended TGFF to generate SystemC models where each thread has a single segment like Figure 2.2a with a base amount of workload. Each segment performs data crunching in a `for` loop like lines 13 and 14 in Figure 2.6b, and the workload is determined by the `for` loop iterations. The base workloads of different segments are generated by TGFF as attributes, randomly distributed in a wide range. Then, in each `while` loop iteration the workload of a segment is adjusted by varying the base value with a random factor, to simulate the variation of execution time due to data dependency. To ensure fair comparison between different dispatchers, each thread defines its own reentrant random number generator (*rand_r()*) to generate the same sequence of random numbers in different simulation runs.

In our experiments, we set the maximum variation of the workload in different iterations to be 0, 20%, 40%, 60%, and 80%. A variation of 0 means the workload of the same segment in different iterations stays the same, a maximum variation of 20% means that the workload in any iteration is within the range of 80% to 120% of the base of the segment, and so on and so forth. In addition, we generate two sets of task graphs that have a different number of parallel threads at each stage (Figure 2.6a), except the first and last stages. The average number of parallel threads per core is chosen in the range of 1 to 2, or 2 to 3. Each set of task graphs contains 30 different benchmarks, and runs on the two workstations.

Table 2.2: Performance of different parallel SystemC schedulers for single-segment threads (Figure 2.2a).

Par	Var	8-Core Host				32-Core Host			
		SEQ	PAR	LJF	SEG	SEQ	PAR	LJF	SEG
1 to 2 threads per core	0	*217s*	513%	**+8.0%**	**+8.0%**	*657s*	1669%	**+20.2%**	+20.1%
	20%	*217s*	501%	**+9.2%**	**+9.2%**	*655s*	1607%	**+17.0%**	+16.9%
	40%	*217s*	480%	**+8.8%**	**+8.8%**	*654s*	1507%	**+12.8%**	**+12.8%**
	60%	*217s*	456%	**+6.8%**	**+6.8%**	*653s*	1407%	**+9.3%**	**+9.3%**
	80%	*217s*	433%	**+5.1%**	**+5.1%**	*652s*	1314%	**+6.5%**	**+6.5%**
2 to 3 threads per core	0	*260s*	574%	**+1.6%**	**+1.6%**	*924s*	1999%	**+11.4%**	**+11.4%**
	20%	*260s*	563%	**+4.4%**	**+4.4%**	*923s*	1937%	**+11.8%**	**+11.8%**
	40%	*259s*	545%	**+4.6%**	**+4.6%**	*921s*	1842%	**+7.9%**	**+7.9%**
	60%	*258s*	526%	+2.5%	**+2.7%**	*920s*	1749%	**+4.3%**	**+4.3%**
	80%	*258s*	510%	**+1.0%**	+0.8%	*918s*	1673%	**+1.8%**	**+1.8%**

Table 2.2 shows the average performance gain of different parallel schedulers over the 30 benchmarks compared with the sequential SystemC simulator from Accellera. The first column *Par* in the table refers to the average number of parallel threads per core at each stage and the second column *Var* refers to the maximum variation of the workload of the same segment in different iterations. For different SystemC schedulers, *SEQ* refers to the sequential SystemC from Accellera, *PAR* refers to our parallel implementation with the Linux scheduling, *LJF* refers to the parallel version with classic LJF dispatching, and *SEG* refers to our segment-aware optmization. The simulation times of *LJF* and *SEG* already include the additional overhead of profiling and sorting. Their relative speedup is compared with *PAR*.

Table 2.2 allows the following observations:

1. **Parallel simulation is fast**: Since the benchmarks have plenty of parallelism inside the models, all parallel simulators achieve a good performance gain on the multi-core hosts, up to 5x on the 8-core, and 20x on the 32-core machine. Also, a larger number of parallel threads leads to higher speedup.

2. ***LJF* and *SEG* are faster than *PAR***: When each thread contains a single segment, the *SEG* scheduler with segment-aware optimization shows the same performance as the classic LJF algorithm. But compared with the parallel simulation that relies on the Linux dispatcher, our segment-aware optimization is clearly better. Also, the segment-aware scheduler achieves greater speedup on the 32-core host than the 8-core host for the same type of benchmarks, as a larger number of processing cores leads to greater variability in thread dispatching.

3. **Prediction needs to be accurate**: In Table 2.2, it is clear that the smaller the variation of the workload in different iterations, the higher the performance gain is. In the case that the maximum variation of the workload is 80% (which is rare in

real world), all the parallel schedulers have similar performance, as the prediction is inaccurate in *LJF* and *SEG*. However, when the maximum variation of the workload is 40%, the *LJF* and *SEG* schedulers still achieve an additional speedup of 8% on the 8-core and 13% on the 32-core host, in the case that the average number of parallel threads per core is in the range of 1 to 2.

2.5.1.2 Multi-Segment Threads

Next, we evaluate our parallel schedulers with benchmarks where each thread has three consecutive segments, like Figure 2.2b. Now in Figure 2.6a, each block in *DUT* still represents an `SC_THREAD`, and it contains three segments that are separated by `wait` statements. The workload in adjacent segments is unrelated and varies independently. The experimental results are shown in Table 2.3.

Table 2.3: Performance of different parallel SystemC schedulers for multi-segment threads (Figure 2.2b).

Par	Var	8-Core Host				32-Core Host			
		SEQ	PAR	LJF	SEG	SEQ	PAR	LJF	SEG
1 to 2 threads per core	0	*220s*	515%	-3.3%	**+6.0%**	*772s*	1634%	+8.0%	**+21.6%**
	20%	*219s*	504%	-1.6%	**+7.7%**	*770s*	1587%	+7.4%	**+16.8%**
	40%	*218s*	482%	+0.8%	**+8.5%**	*768s*	1484%	+6.9%	**+12.9%**
	60%	*217s*	459%	+2.4%	**+7.2%**	*767s*	1380%	+6.1%	**+9.7%**
	80%	*216s*	437%	+3.2%	**+5.7%**	*765s*	1290%	+5.0%	**+6.6%**
2 to 3 threads per core	0	*263s*	564%	-2.0%	**+3.2%**	*829s*	1952%	+0.8%	**+13.5%**
	20%	*262s*	555%	-1.3%	**+5.8%**	*828s*	1899%	+0.6%	**+13.8%**
	40%	*261s*	539%	-0.7%	**+5.8%**	*826s*	1818%	+0.2%	**+9.7%**
	60%	*261s*	522%	-0.4%	**+3.6%**	*824s*	1736%	+0.6%	**+5.4%**
	80%	*260s*	507%	-0.8%	**+1.8%**	*822s*	1657%	+1.0%	**+3.2%**

In addition to the first three observations from Table 2.2, we make another observation in Table 2.3:

4. **Segment-awareness matters**: In contrast to Table 2.2, the performance gain of *LJF* degrades because the prediction of segment run time is inaccurate. On the other hand, the segment-aware scheduler achieves a significant speedup over the other two parallel schedulers. For example, on the 32-core host, when the number of parallel threads per core is within the range of 1 to 2 and the workload variation is 0, the three parallel schedulers speed up by 16x to 19x. Compared with the parallel scheduler with Linux dispatching, the segment-aware optimization achieves another 20% speedup (while the relative improvement of LJF is less than 10%).

2.5.1.3 General Threads

Finally we evaluate our parallel schedulers with benchmarks in which each thread has multiple segments in a general structure, as Figure 2.2c. Table 2.4 shows the experimental results and allows another observation:

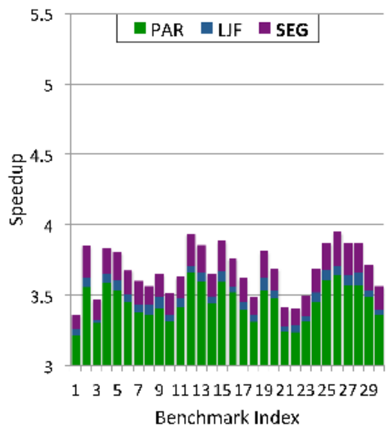
5. **Segment-aware scheduler identifies the correct segments**: In Table 2.4, even though a thread may take different execution paths, our segment-aware scheduler still identifies the next segment correctly and achieves a high speedup over the other two parallel schedulers (more than 35% over *PAR* on the 32-core host, when the number of parallel threads per core is 2 to 3 and the workload variation is 0).

However, the performance of the three parallel schedulers is similar on both hosts when the parallelism is low (1 to 2 threads per core). This is due to the fact that each thread takes different execution paths and has a different number of segments in total. Thus, at a certain point in the simulation, most threads finish all their segments in the current iteration but some threads have extra segments to execute in the following delta cycles. That reduces the parallelism in the simulation (i.e. the number of parallel threads is smaller than the number

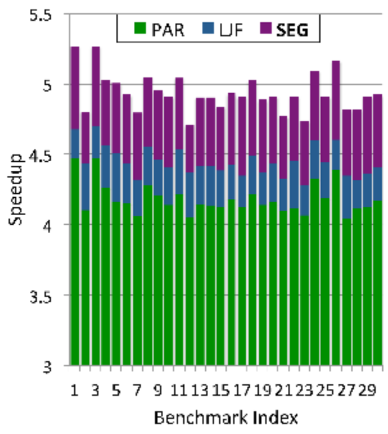
Table 2.4: Performance of different parallel SystemC schedulers for general threads (Figure 2.2c).

Par	Var	8-Core Host				32-Core Host			
		SEQ	PAR	LJF	SEG	SEQ	PAR	LJF	SEG
1 to 2 threads per core	0	*183s*	370%	+2.4%	**+6.2%**	*402s*	844%	**+4.1%**	+3.4%
	20%	*183s*	362%	+2.2%	**+6.9%**	*402s*	807%	+2.5%	**+2.9%**
	40%	*183s*	345%	+1.7%	**+6.7%**	*401s*	750%	+0.4%	**+3.6%**
	60%	*183s*	326%	+1.5%	**+6.1%**	*401s*	692%	+0.7%	**+2.3%**
	80%	*183s*	308%	+1.3%	**+5.5%**	*401s*	643%	-1.4%	**+3.1%**
2 to 3 threads per core	0	*218s*	439%	+6.8%	**+21.0%**	*572s*	1201%	+19.2%	**+37.0%**
	20%	*218s*	433%	+6.5%	**+19.9%**	*572s*	1183%	+20.9%	**+35.2%**
	40%	*218s*	418%	+6.5%	**+18.2%**	*571s*	1121%	+18.6%	**+31.7%**
	60%	*218s*	401%	+6.0%	**+15.7%**	*571s*	1056%	+17.0%	**+28.8%**
	80%	*217s*	384%	+5.2%	**+13.5%**	*570s*	1008%	+15.2%	**+23.7%**

of cores), in which case the classic LJF dispatcher and our segment-aware optimization perform the same as the Linux dispatcher. The performance of the segment-aware scheduler improves a lot when the parallelism increases to 2 to 3 threads per core.



(a) 1 to 2 threads per core.



(b) 2 to 3 threads per core.

Figure 2.7: Performance comparison for general threads (Figure 2.2c) on a 8-core host.

Figure 2.7 shows the performance of different parallel schedulers for each individual benchmark on the 8-core host. Again, each thread contains multiple segments in a general structure and the maximum variation of the workload is 40%. The number of parallel threads per core for Figure 2.7a is within 1 to 2, and Figure 2.7b has 2 to 3 parallel threads per core. Here, we make another observation:

6. **Our segment-aware scheduler consistently shows the best performance**: For all 60 benchmarks, even though they have different segment graphs, our segment-aware scheduler always achieves the highest speedup, significantly better than the other two parallel schedulers.

2.5.2 Canny Edge Detector Example

For a first real-world experiment, we use a pipelined Canny edge detector to demonstrate the performance gain of our segment-aware optimization. The Canny edge detector is a popular image processing application to detect a wide range of edges in images. Figure 2.8 shows the block diagram of the Canny edge detector in SystemC. In this model, seven functions (i.e. *Prep*, *BlurX_Par*, *BlurY_Par*, *Derivative_x_y*, *Magnitude_x_y*, *Non_Max_Supp* and *Apply_Hysteresis*) are applied to a sequence of input images in a pipelined fashion. In *BlurX_Par* and *BlurY_Par*, multiple parallel threads work on different slices of the image. The number of parallel threads in these two modules is configurable as an exponent of 2. In Figure 2.8, all blocks are implemented as `SC_MODULE` and communicate through `sc_fifo` channels. The parallel modules in the pipeline may be blocked by the `sc_fifo` channels, and have multiple segments in one thread. Thus, their segment structure is similar to that in Figure 2.2c. The execution of some segments is optional, depending on whether the buffers in the `sc_fifo` channels are empty or not. As a result, the LJF algorithm degrades due to inaccurate predictions, whereas our segment-aware optimization achieves a much better performance.

Table 2.5 shows the performance of different parallel SystemC schedulers for the pipelined Canny edge detector example on the 8-core host. Here, the relative speedup of *LJF* and

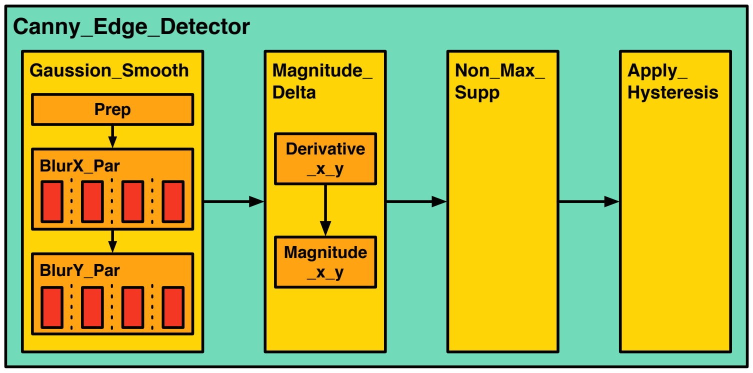


Figure 2.8: Pipelined Canny edge detector example.

Table 2.5: Performance comparison for Canny edge detector on a 8-core host.

Benchmark	SEQ	PAR	LJF		SEG		
	Time	Speedup	Speedup	+Speedup	Speedup	+Speedup	Overhead
canny_v1	*138.8s*	**298.1%**	290.4%	-2.6%	284.6%	-4.5%	0.05%
canny_v2	*139.0s*	305.7%	296.4%	-3.1%	**360.4%**	+17.9%	0.15%
canny_v3	*139.0s*	255.9%	261.8%	+2.3%	**329.9%**	+28.9%	1.75%

SEG is compared with *PAR*, and *canny_v1*, *canny_v2* and *canny_v3* have 1, 8 and 256 worker threads in *BlurX_Par* and *BlurY_Par* respectively[4]. Clearly, the LJF scheduler has similar or worse performance than the default Linux scheduler, but our segment-aware algorithm achieves an additional speedup of up to 28%. Only when the number of parallel threads in the model is lower than the number of cores on the host (e.g. *canny_v1* has up to 7 parallel threads, made up of the seven pipelined stages in the model), *LJF* and *SEG* are slightly worse due to the small profiling and scheduling overhead.

2.5.3 JPEG Encoder Example

Table 2.6: Performance comparison for JPEG encoder on a 8-core host.

Benchmark	SEQ	PAR	LJF		SEG		
	Time	Speedup	Speedup	+Speedup	Speedup	+Speedup	Overhead
JPEG	*176.1s*	331.4%	324.7%	-2.0%	**400.6%**	+20.9%	0.87%

[4]As these three benchmarks process the same sequence of images, a larger number of parallel threads means a smaller amount of workload in each worker thread.

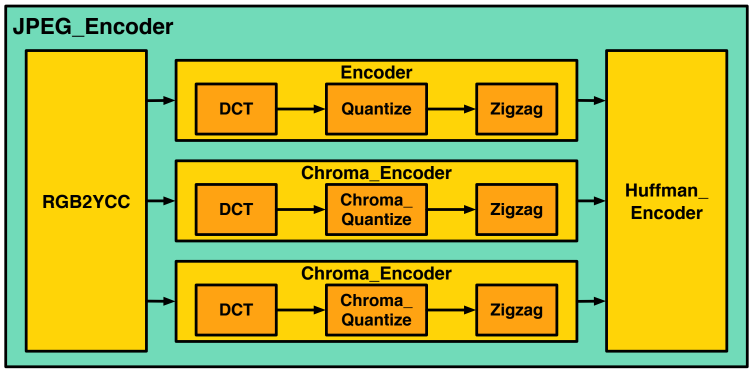


Figure 2.9: Pipelined JPEG encoder example.

Our second real-world experiment uses a JPEG image encoder (an extended version of [10]). Its block diagram is shown in Figure 2.9. Here, the *RGB2YCC* module first performs color-space transformation on an image from RGB to YCbCr. Then, the image is split into blocks of 8×8 pixels and each color component (Y, Cb or Cr) of a block undergoes Discrete Cosine Transform (DCT), quantization, and zigzag ordering separately. At the end, the resulting data for all 8×8 blocks is further compressed with the lossless Huffman encoding algorithm. Since encoding of the three color components (Y, Cb and Cr) is independent, the JPEG model executes these three encoders in parallel. Also, to efficiently process a stream of images, our JPEG encoder example is implemented in a pipelined fashion. Same as the Canny edge detector, each block in Figure 2.9 is implemented as an `SC_MODULE` and using `sc_fifo` for communication. Thus, each thread has multiple segments and owns a segment structure like Figure 2.2c.

Table 2.6 compares the performance of different SystemC schedulers for the JPEG encoder example on the 8-core host. Again, our segment-aware optimization shows the best performance, achieving an additional 20% speedup over *PAR*. In comparison, the LJF scheduler is slightly worse than the Linux scheduler (*PAR*) due to the inaccurate prediction based on previous run times.

Table 2.5 and 2.6 also show the profiling and sorting overhead of our segment-aware optimization. Clearly, the overhead of our proposed algorithm takes less than 2% of the whole simulation time, and sometimes much less (e.g. 0.05% and 0.15% in the cases of *canny_v1* and *canny_v2*). In order to keep the per-segment execution time information, the extra storage overhead is two `unsigned long long` values (start time stamp and previous run time) per segment and one `unsigned int` value (current segment ID) per thread.

2.6 Conclusion

In this chapter, we proposed a segment-aware scheduling algorithm [57] with optimized thread dispatching in the context of a parallel SystemC simulator. By taking the execution time for a specific segment as a prediction of the next run time, our approach dispatches threads in an optimized and efficient fashion. Evaluated with synthetic benchmarks and real-world examples, the implemented parallel simulator shows a speedup of up to 20x over the sequential simulator. More importantly, our segment-aware optimization works on top of this and consistently achieves a high speedup over previous thread dispatch algorithms for examples with complex segment structures. Based on these experimental results, we conclude that accurate prediction of the next execution time based on segment information is critical. Our segment-aware approach achieves significantly better performance than previous schedulers.

Chapter 3

Core Distance

In addition to load-balancing optimization in Chapter 2, we investigate communication minimization in this chapter. We first define the concept of *core distance* [58] for multi-core and many-core platforms, and then propose an algorithm to optimize thread-to-core mapping in order to minimize on-chip communication overhead.

3.1 Introduction

Many-core processors have become popular in recent years to provide capable platforms for those highly parallel applications which have extraordinary scaling and vector capabilities that cannot be satisfied by conventional multi-core processors. Generally speaking, many-core platforms refer to processors with dozens to hundreds and soon thousands of cores on chip. Thus, in order to fully utilize many-core platforms, an application must scale well past hundreds of threads and distribute equal workloads among those parallel threads.

One recent example of a many-core processor is the Xeon Phi coprocessor, a readily available implementation of the Intel Many Integrated Core (MIC) architecture. Figure 3.1 depicts

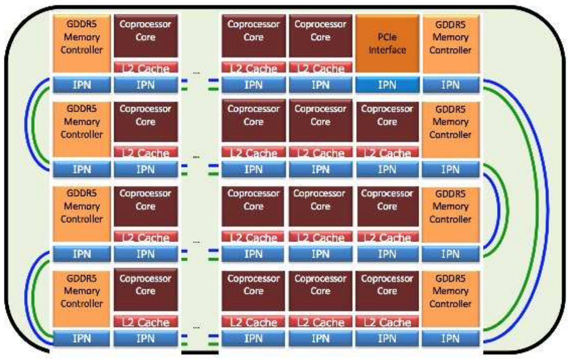


Figure 3.1: Intel Xeon Phi coprocessor architecture [39].

the conceptual structure of the Intel Xeon Phi architecture [39]. On the single die of the coprocessor, up to 61 x86-based cores are integrated. These parallel processing cores communicate via a high performance bidirectional ring interconnect. Each core is fully functional and fetches and decodes instructions in-order from four hardware thread contexts. Thus, in total, there are 240 logical cores available on the Intel Xeon Phi 5110P coprocessor. For the on-chip cache hierarchy, each core includes 32 KB L1 instruction and data cache, as well as a private 512 KB L2 cache. In order to keep the L2 cache data globally consistent and reduce hot-spot contention for data references, a distributed Tag Directory (TD) is coresident with each core to cross-snoop L2 caches in all cores. Every physical memory address is uniquely mapped to one of 64 distributed tag directories on the ring network , which is not necessarily co-located with the core generating the cache miss. This mapping is accomplished through a reversible hash function. Using the distributed TD infrastructure, the caches are kept consistent without software intervention. In addition to the individual core and distributed tag directory, the coprocessor includes up to eight memory controllers supporting two GDDR5 memory channels, and a PCI Express system I/O logic connecting the host Intel Xeon processor. In general, the Xeon Phi coprocessor can be viewed as a symmetric multiprocessor (SMP) with shared Uniform Memory Access (UMA) [40][42].

3.1.1 Motivation

Increasing the number of cores provides potential to improve performance and scalability for highly parallel programs, but also brings downsides. As the chip size is enlarged to accomodate more processing units, core-to-core transfers are not always significantly better than main memory access and optimization becomes crucial. For the Xeon Phi coprocessor specifically, because of the opaque hashing method and the resulting "random" distribution of addresses around the ring, no previous software optimization has been found to improve core-to-core transfers significantly [42]. In the remainder of this chapter, we propose our software strategy to optimize thread-to-core mapping on many-core platforms with distributed tag directories, and show that it minimizes core-to-core communication latency.

The rest of the chapter is organized as follows: Section 3.2 first defines the concept of *core distance* on SMP architectures and shows measurements of core distance on Intel Xeon and Intel Xeon Phi processors. In Section 3.3, based on our observations, we propose our approach to optimize thread-to-core mapping and speedup core-to-core transfers for many-core platforms which use distributed directory-based cache coherence protocols. Next, Section 3.4 provides experimental results for two communication-intensive benchmarks to demonstrate the performance improvement of our optimization approach. Finally, relevant related work is given in Section 3.5 and we conclude our work in Section 3.6.

3.2 Core Distance

In this section, we define core distance and provide a memory ping-pong benchmark to quantify it. We then show measured core distances for a multi-core and many-core processor to show the architectural differences between these platforms.

3.2.1 Definition of Core Distance

On current multi-core and many-core systems, each core usually has its own local cache to utilize program locality and speed up memory access. In order to keep the data globally coherent among all caches, the modification of a data block is broadcast on the interconnecting medium (in snooping protocols) or passed to the directory that tracks the state of the cache (in directory-based protocols). By maintaining coherent caches, data can be easily transferred via shared variables between cores. However the core-to-core transfer latency varies a lot between cores on the same processor and different processors. To quantify the core-to-core tranfer latency, we can define *core distance* as the duration it takes to move a data word back and forth between two cores (one round trip). A larger core distance then means that it is more expensive to share data between these two cores.

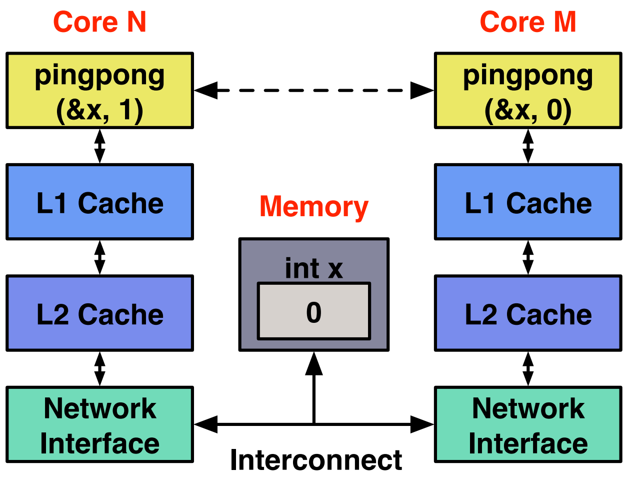


Figure 3.2: Ping-pong communication for measuring core distance.

In order to measure core distances on multi-core and many-core processors, we propose a memory ping-pong communication benchmark. Figure 3.2 shows the communication between two cores and Algorithm 3 lists the *pingpong* function executing in the threads, which are bound to the measured cores. Each thread starts a timer in local variable T_1. Then, it compares the shared memory address *varptr* against its local value *val*. If they match, the program goes into the `while` loop, pauses the core for a few clock cycles and then checks

Algorithm 3 Ping-Pong Communication Function

```
1: **cycles_t** pingpong ($varptr$, $val$) {
2: $T_1 \leftarrow$ CurrentCycles()
3: **for** $iter = 1$ to ITERATION **do**
4:     **while** Load($varptr$) $= val$ **do**
5:         Processor Pause
6:     **end while**
7:     Store($varptr$, $val$)
8: **end for**
9: $T_2 \leftarrow$ CurrentCycles()
10: **return** $(T_2 - T_1)$/ITERATION
11: }
```

the sharing address again. Otherwise, the thread stores its *val* to the sharing address which is then communicated through the cache hierarchy to the other core. As the shared data block is in the cache of both cores, the new value in the sharing address will transfer through interconnection path to the other core to maintain cache coherence. Each thread tries to update the shared variable in turn and runs this procedure for multiple iterations. At the end, the program stops the timer, and returns $(T_2 - T_1)$/ITERATION as the average round-trip communication latency.

3.2.2 Core Distances on Hierarchical Multi-Core Platforms

As expected, on multi-core systems using snooping cache coherency, core distance is highly correlative to the core placement on the platform. For example, Figure 3.3 illustrates the architecture of a Intel Xeon dual-CPU system. Figure 3.4 shows the corresponding measured core distances from Core 0 to other cores. As hyperthreading is enabled, there are 32 logical cores in total on this platform. In Figure 3.4, the core distance between the two logical threads on the same physical core (Core 0 and 16) is minimum as the two hyperthreads can communicate via the local L1 or L2 cache. Since Core 0 to 7 are on the same processor and Core 8 to 15 are on the other one, interprocessor communication is more expensive than intraprocessor communication, costing 1200 vs. 300 cycles for a round trip. As Core 17 to 31 are hyperthreads of Core 1 to 15, the core distances from Core 0 to these cores are repeated[1].

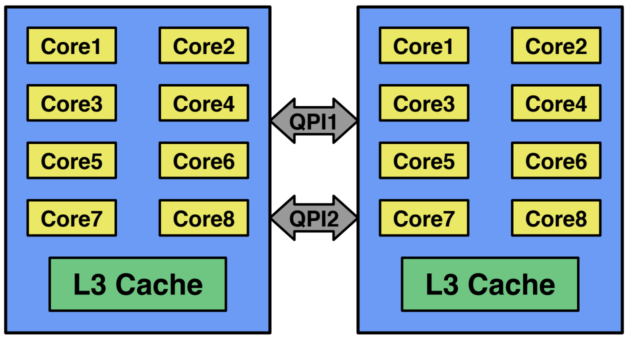


Figure 3.3: Intel Xeon E5-2680 dual-CPU architecture.

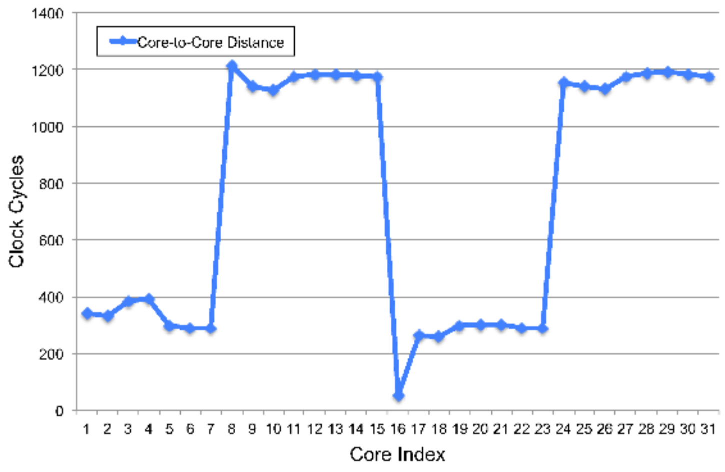


Figure 3.4: Core Distances from Core 0 to other cores on Intel Xeon E5-2680.

3.2.3 Core Distances on Many-Core Platforms

We see that core distance is not always correlating with the physical distance on chip. For a many-core system communicating over a ring network, such as the Xeon Phi 5110P coprocessor, one could expect the core distance as indicated with the green line in Figure 3.5, where Core 0 has a shorter core distance to its neighbors than to the opposite core on the ring

[1]Core distance is symmetric between two cores.

network. Specifically, this chip contains 60 physical cores connected by a high performance bidirectional ring network. As each core can fetch instructions from four hardware contexts, there are 240 logical cores available. Every four consecutive cores (starting from 1, say 1/2/3/4, 5/6/7/8, ..., 237/238/239/0) denote four hardware threads on the same physical core. Adjacent cores are placed next to each other in a ring topology. In contrast to the expectation, the measured core distances from Core 0 to Core $4n+1 (0 \leq n \leq 59)$ are shown in the blue line in Figure 3.5. Except for Core 237, which is another hardware thread on the same physical core, Core 125 exhibits the shortest core distance to Core 0.

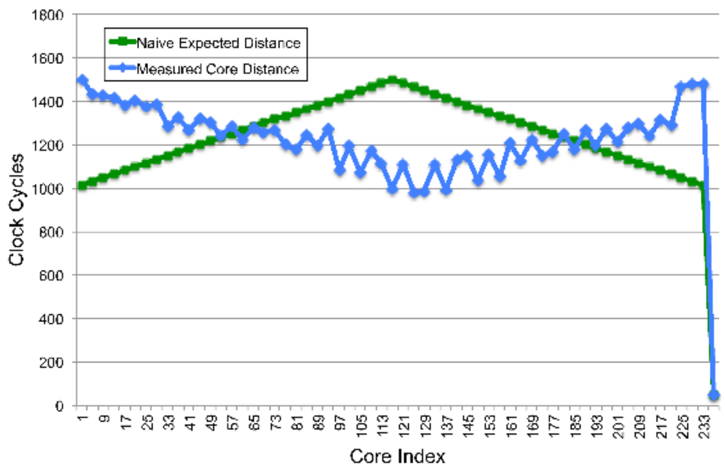


Figure 3.5: Core distances from Core 0 to other cores on Xeon Phi 5110P.

On a closer look, we note that the Intel Xeon Phi coprocessor uses a directory-based cache coherence protocol. The distributed tag directory maintaining cache coherence plays an important role in the core-to-core communication. Figure 3.6 depicts the detailed communication model on this platform. When one thread (**Pong** in Figure 3.6) has a cache miss and needs to fetch an updated value, it first talks to the responsible tag directory (the red TD in Figure 3.6) to find which core cache contains the new value (Step 1). Every memory reference the processor generates is mapped through a one-to-one hash function to a TD based on the physical address. Notably, the responsible TD is not necessarily co-located with the core generating the cache miss [40] and could be associated with any core on the chip. When the responsible TD finds another core (**Ping** in Figure 3.6) owns the updated value, it sends a request for the new value to the specific core (Step 2), gets the value from that core (Step 3) and passes it back to the core exeperiencing the cache miss (Step 4). Finally, the TD updates the sharing status of the cache block in the first core. Also, when one core writes to the shared data block, it needs to notify the tag directory to invalidate the sharing value in other cores. The large variation in the measured core-to-core communication latency in Figure 3.5 [2], i.e., several hundred cycles, motivates the idea in this chapter. If we place communicating threads close to the TD in question, we can speed up this communication significantly.

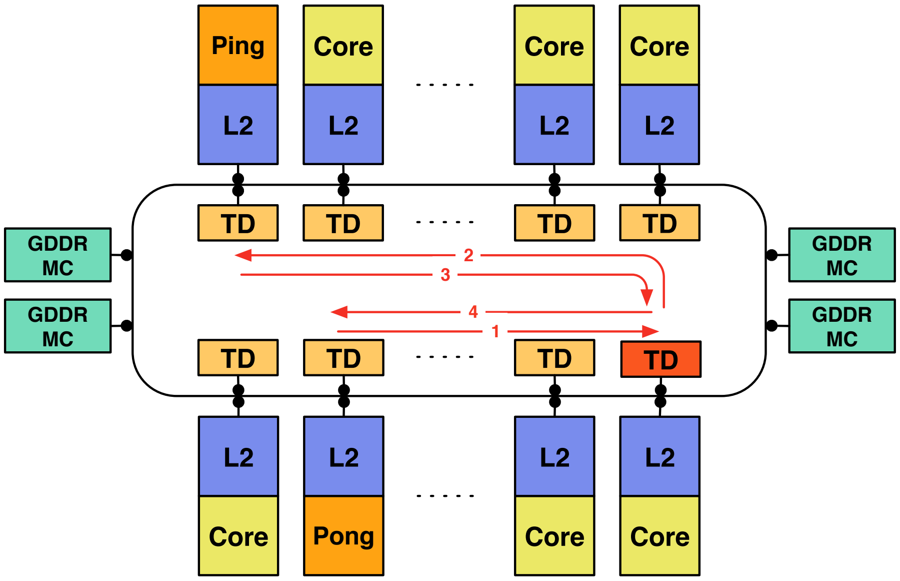


Figure 3.6: Cache coherence via distributed tag directories (TD).

3.2.4 Core Distances on Busy Many-Core Platforms

When multiple pairs of communication happen concurrently, the core distances on the many-core processor become even more expensive and unpredictable. Figure 3.7 shows the core distances from Core 0 to Core $4n + 1 (0 \leq n \leq 59)$ on a busy Xeon Phi coprocessor (green line), in comparison to the situation when only two cores communicate with each other on the

[2] At this point, we cannot explain the "zig-zag" behavior among neighboring cores. We suspect this stems from other bus traffic and/or Linux interference on the ring network.

chip (blue line, same as the blue line in Figure 3.5). Here we run a Monte Carlo simulation which issues 60 pairs of concurrent communications and randomly distributes them on the ring network. As half of the chip is busy and the cores compete for the access to network, the core-to-core communication latency grows up to 10,000 cycles for one round trip. Also, the location of the responsible tag directory becomes negligible compared to the interference between parallel communications.

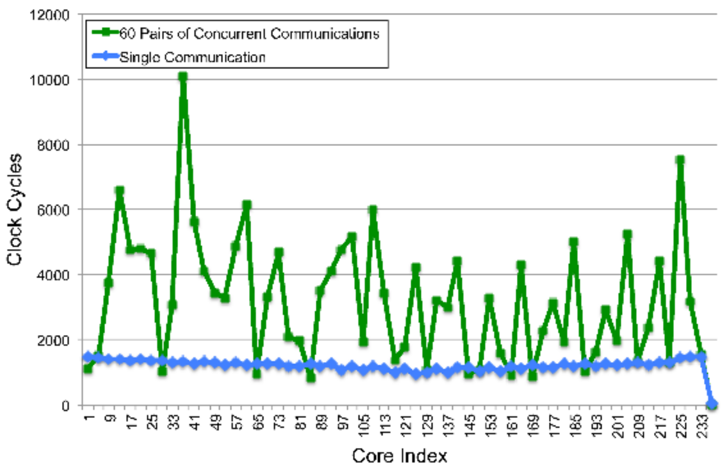


Figure 3.7: Core distances from Core 0 to other cores at 50% core utilization.

3.3 Minimizing Inter-Core Communication

Many-core processors provide a high performance of parallel computing, but the long communication latency (more than 1000 cycles for one round-trip core-to-core communication) may slow down the execution. In this section, we exploit the above observations and improve application performance by optimizing the thread-to-core mapping to minimize the on-chip communication overhead.

3.3.1 Mapping Communicating Threads Close to the TD

Based on the core distance curve in Figure 3.5, we can infer the location of the responsible TD by finding the core index which has the shortest core distance, ignoring the logical cores on the same physical core. Algorithm 4 describes the function to find the responsible tag directory on the ring network. In Function *FindTD*, one thread is fixed to Core 0 and the other thread is set to the other cores on the ring network. Both threads run the *pingpong* function simultaneously. As one thread may start first and suspend for the other, we choose to use the core distance measured in the thread starting later (by checking timestamp T_1 of each thread). The function will find the minimum core distance and return the corresponding *coreindex* as the location of the identified tag directory.

Algorithm 4 Find the Responsible Tag Directory

```
1: unsigned int FindTD (char $*var$) {
2: for all $c \in$ ProcessorCores do
3:     set core affinity to 0 for Thread $th_1$
4:     set core affinity to $c$ for Thread $th_2$
5:     create Thread $th_1$ to run $t_1 \leftarrow$ pingpong($var, 0$)
6:     create Thread $th_2$ to run $t_2 \leftarrow$ pingpong($var, c$)
7:     if $th_2$ starts later then
8:         $coredist_{0,c} \leftarrow t_2$
9:     else
10:        $coredist_{0,c} \leftarrow t_1$
11:    end if
12: end for
13: return $coreindex$ where $coredist_{0,coreindex} = \min\{coredist_{0,c}\}$
14: }
```

With the knowledge of the TD location we can reduce the communication latency. By invoking the *FindTD* function at the beginning of the program, our approach profiles the application and determines the TD locations of the shared variables[3] in the application. Mapping threads close to these tag directories will reduce the onchip communication over-

[3]The size of a shared variable should be smaller than one cache block (64 bytes on Intel Xeon Phi) and mapped to one tag directory.

head[4]. Algorithm 5 and Algorithm 6 compare the thread initialization with and without optimization respectively. Rather than using the default thread attributes in Algorithm 5, Algorithm 6 first finds the tag directories td_i of each communication channel ch_i, and then sets the core affinity of the communicating thread close to td_i. *GetCore* returns a core index which is near td_i and tries to balance work load among all available cores in a greedy fashion. Next, all communicating threads are created with the affinity-optimized thread attributes.

Algorithm 5 Thread Initialization without Optimization

```
1: for all Thread $th_{i,1}, th_{i,2}$ using Channel $ch_i$ do
2:     $th_{i,1}$.ThreadCreate($DefaultThreadAttributes$)
3:     $th_{i,2}$.ThreadCreate($DefaultThreadAttributes$)
4: end for
```

Algorithm 6 Thread Initialization with Optimization

```
1: for all Channel $ch_i$ do
2:     $td_i \leftarrow$ FindTD(addressof($ch_i$))
3:     set core affinity to GetCore($td_i$) in $ThreadAttributes_{i,1}$
4:     set core affinity to GetCore($td_i$) in $ThreadAttributes_{i,2}$
5: end for
6: for all Thread $th_{i,1}, th_{i,2}$ using Channel $ch_i$ do
7:     $th_{i,1}$.ThreadCreate($ThreadAttributes_{i,1}$)
8:     $th_{i,2}$.ThreadCreate($ThreadAttributes_{i,2}$)
9: end for
```

Figure 3.8 shows the optimized core distances for the Intel Xeon Phi coprocessor, with one thread placed at the responsible TD (Core 125, using the same TD as Figure 3.5). The minimum core distance here (except the logical core on the same physical hardware) is about 500 cycles, when mapping the other thread next to the tag directory. This is much lower than the core distance in Figure 3.5 which is as high as 1500 cycles. Also, the core distance is now *predictable* with the physical location of the core, as indicated by the blue line in Figure 3.8. The further two cores are on the ring network, the larger the core distance is between them.

[4]As the hash function is unknown and the mapping is based on the physical address of the memory reference, it is very difficult to allocate a shared variable whose responsible tag directory is close to the communicating cores. So instead of moving the TD, we move the threads.

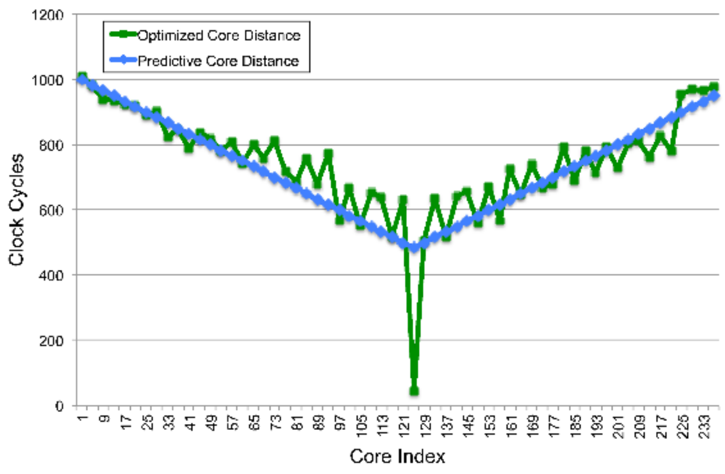


Figure 3.8: Optimized inter-core communication latency.

3.3.2 Pizza Slice Algorithm

In a busy situation where many pairs of core-to-core communication run simultaneously, transfer latency is extremely expensive (Figure 3.7) and unpredictable. Here, in order to reduce the interference and improve onchip communication, we propose a software algorithm which divides the ring network into a given number of sections and localizes the communication into these sections. We name this the *Pizza Slice Algorithm* due to its similarity with a sliced pizza (Figure 3.9). By evenly distributing the concurrent communications onto a given number of different slices of the ring network, our approach maps threads onto cores of one section and selects a shared variable whose responsible tag directory is in the same section[5].

Figure 3.10 compares the core distances from Core 0 to Core $n (1 \leq n \leq 39)$ in one section, before and after applying the *Pizza Slice Algorithm* to the Xeon Phi coprocessor in which half of the chip is busy communicating. Core distances are measured using Monte Carlo simulation in both cases. Before applying the Pizza Algorithm, threads are randomly distributed on the ring network and the responsible tag directories might be in different section from

[5]We need to allocate a few (about 25%) more shared variables than defined in the program so as to find enough communication channels in each section.

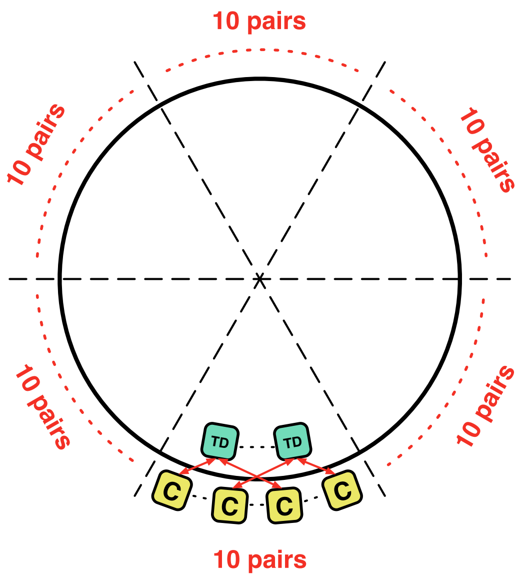


Figure 3.9: ”Pizza Slice Algorithm” for cores and TDs on a ring network.

communicating cores. With the *Pizza Slice Algorithm*, the threads and tag directories are localized into one section, and communication latencies decrease dramatically (from more than 7000 to less than 1000 cycles).

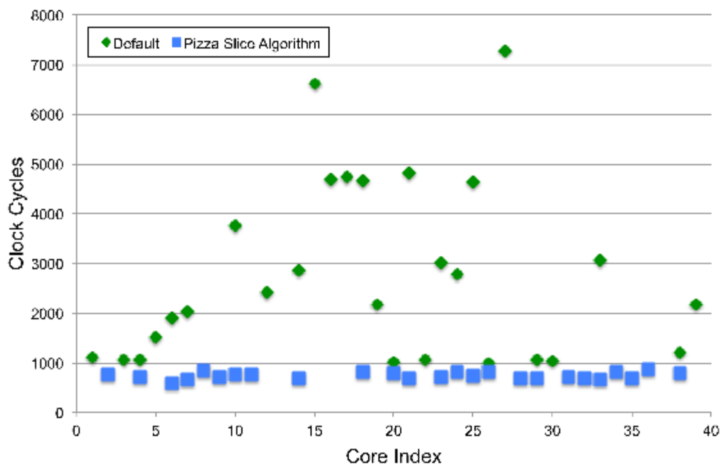


Figure 3.10: Comparison of communication latency before and after applying the Pizza Slice Algorithm.

3.4 Experimental Results

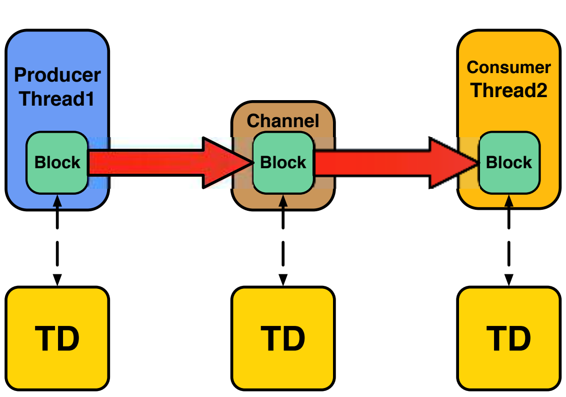
To demonstrate the performance speedup for actual applications, we present experimental results of two communication-intensive benchmarks: a producer-consumer example and a Mandelbrot set renderer model. All the experiments are conducted on the Intel Xeon Phi 5110P coprocessor running at 1.052 GHz.

3.4.1 Producer-Consumer Model

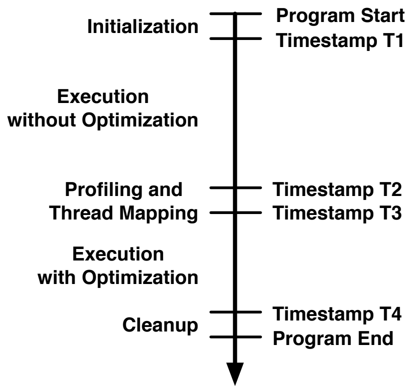
Our producer-consumer model is a classic example of a multiprocess synchronization problem. Figure 3.11a shows the block diagram where Producer and Consumer are children of the root thread, communicating a data value through a spin-locked channel[6] which has a buffer size of one. The Producer generates a data value and puts it into the channel buffer. After the Consumer fetches the data, the Producer resumes to generate new value. In addition to the channel buffer, the Producer and Consumer respectively own a local variable to store the communicated data. The channel buffer is accessed by both Producer and Consumer threads, which contain a copy of the data block in their local caches. Thus, the responsible tag directory for the channel data is of major concern in the communication. To speed up the synchronization, we can map Producer and Consumer threads close to the tag directory of the channel block.

To ensure a fair comparison, we run both the default and our optimized version in the same process, so that the same TDs are used. Figure 3.11b shows the timeline of our experimental evaluation. First, the model runs with Linux default settings (from $T1$ to $T2$). Next, we profile the application and apply our optimization. The overhead of profiling and optimizing the thread mapping is measured by $T3-T2$. Finally, the model runs again with the optimized

[6]Rather than using POSIX threads API to suspend and resume threads, spin-locked channels eliminate time variation due to Linux kernel execution.



(a) Block diagram.



(b) Execution timeline.

Figure 3.11: Producer-consumer example.

thread-to-core mapping from $T3$ to $T4$. The performance speedup of the model is calculated by dividing $T2 - T1$ by $T4 - T2$ (i.e., including the profiling overhead).

3.4.1.1 Assigning Threads onto One Physical Core

According to Figure 3.5, communication between logical threads on the same physical core is the fastest possible option. Figure 3.12 shows the performance improvement of the producer-consumer model by mapping both threads to the same physical core. Running the benchmark for 100 iterations, we show the statistical results of the application speedup.

As the inner core distance is 10 times smaller than that between physical cores, the optimized producer-consumer model achieves an order of magnitude higher execution speed (7x to 16x) than the original model. The variation of the performance speedup is due to the varied locations of the responsible TD and resulting execution time in the original model. On average the performance increases by 11.66x.

Figure 3.13 shows the histogram of the performance speedup. The shape resembles a Gaussian distribution with its mean value around 11x.

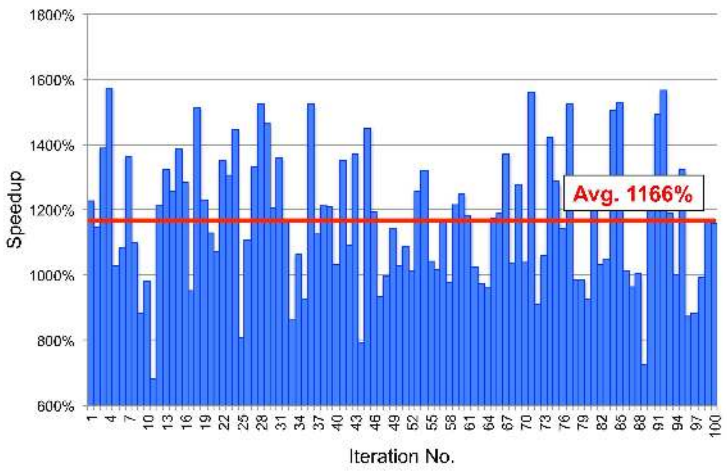


Figure 3.12: Speedup of the producer-consumer example when mapping threads onto the same physical core.

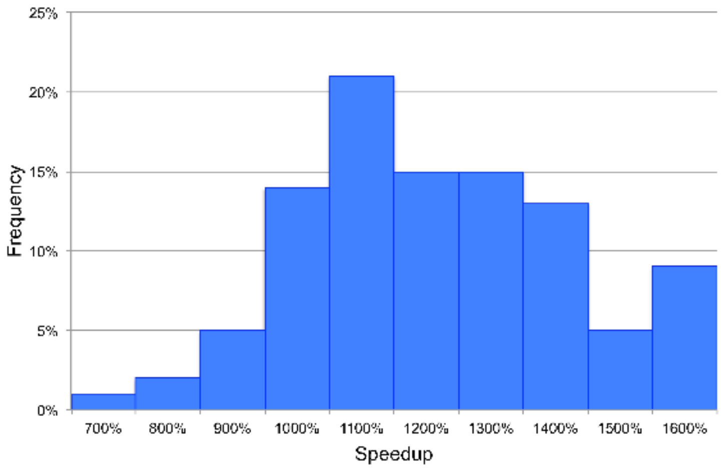


Figure 3.13: Histogram of the producer-consumer example speedup when mapping threads onto the same physical core.

3.4.1.2 Assigning Threads to Close Cores

In most applications, it is a bad idea to map threads onto the same physical core. Figure 3.14 shows the performance of the producer-consumer example when assigning the threads onto cores close to the TD (one core co-located with the tag directory, and the other next to it). Here, our approach shortens communication latency and gains speedup in most experiments, up to 2.5x. In some rare cases (2 of the 100 iterations), if the Linux default scheduler "luckily" sets threads to the optimal cores, our approach performs only a little worse (95%) due to our overhead. Figure 3.15 reflects the histogram of the performance speedup in Figure 3.14. The mean value of the speedup is 1.45x.

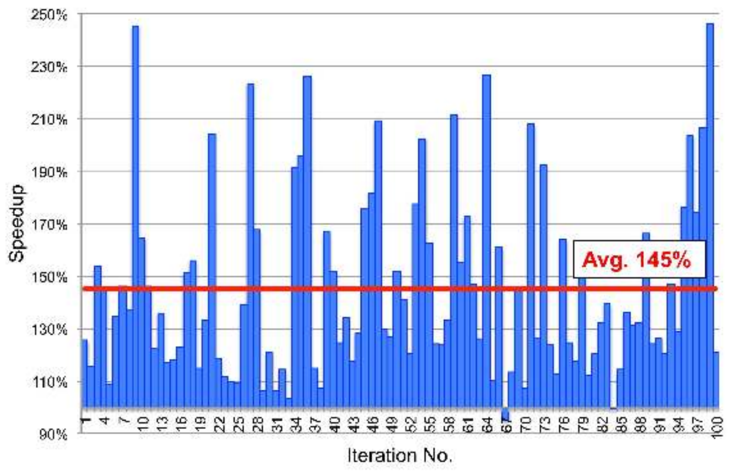


Figure 3.14: Speedup of the producer-consumer example when mapping threads close to the TD.

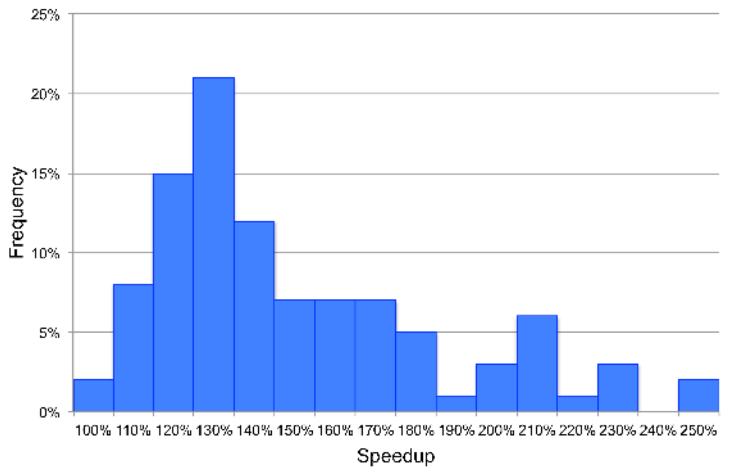


Figure 3.15: Histogram of the Producer-Consumer Model Speedup by Mapping Threads Close to the TD.

3.4.2 Mandelbrot Set Renderer Example

As an example of highly parallel graphics applications, we use a renderer for Mandelbrot set images [61]. Figure 3.16 shows its block diagram. In the model, there are four Coordinators which work on four separate slices of each frame and send coordinates in the complex plane to 8 Mandelbrot worker threads. Each worker thread calculates whether the point belongs to the Mandelbrot set or not. The coordinates of each pixel in the image is sent through a spin-locked channel. Zooming into an interesting area of the image, our application generates a series of 100 Mandelbrot set images.

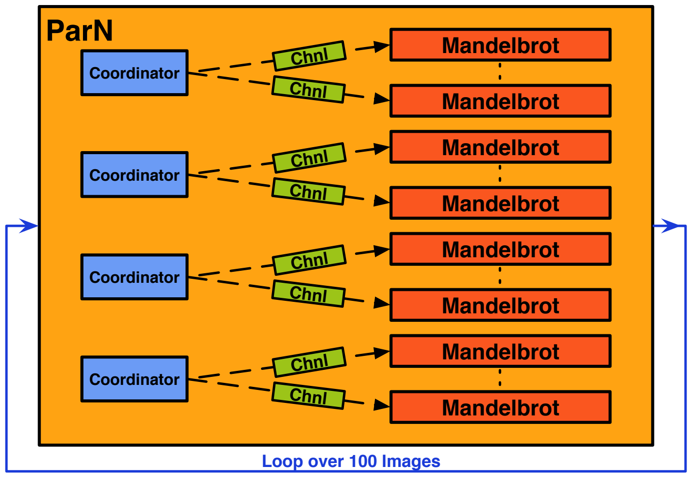
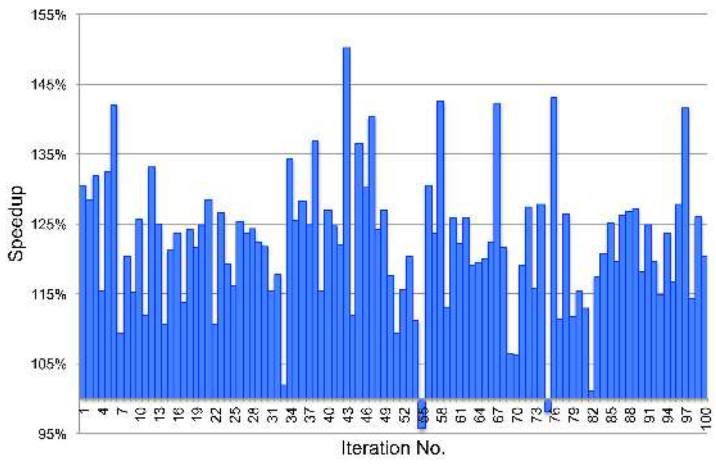
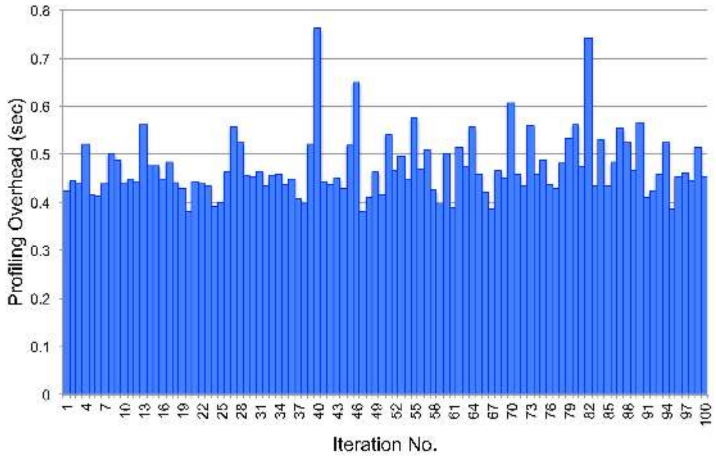


Figure 3.16: Block diagram of the Mandelbrot renderer example.

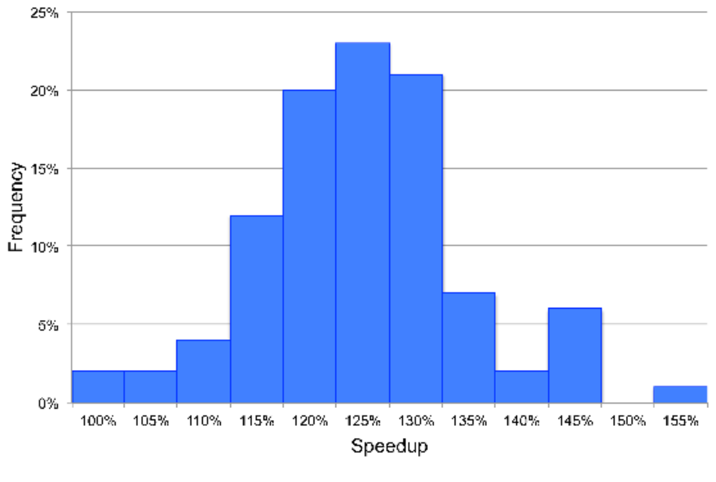
Figure 3.17a shows that our *Pizza Slice Algorithm* accelerates the graphics application by a maximum speedup of 150% and a minimum of 95%. The Pizza approach optimizes the application by mapping each Coordinator and its associated worker threads and channel tag directories into one of four sections on the ring network. Worker threads are assigned to the same cores of the responsible TDs (or next to TD to balance CPU load) and the Coordinator is placed in the middle of the section. Figure 3.17b shows the overhead of profiling and optimization (time $T3 - T2$ in Figure 3.11b). Compared to the execution time of the new model ($T4 - T2$), the profiling overhead is less than 0.8 second absolute, or 1% of the execution time. The speedup is 125% on average, as shown in the histogram in Figure 3.17c.



(a) Performance speedup.



(b) Profiling overhead.



(c) Histogram of the speedup.

Figure 3.17: Apply Pizza Slice Algorithm to the Mandelbrot renderer.

3.5 Related Work

With the fast increasing number of processing cores on a single die, we expect hundreds and even thousands of cores integrated on the Chip Multiprocessor (CMP) systems, which are known as many-core platforms. Currently, some many-core systems are readily available, e.g. Tilera [1], Intel's TeraFlop [89] and MIC [39]. Together with GPGPU and distributed shared memory (DSM) systems, these high-performance computing platforms provide abundant parallel processing power. How to efficiently exploit these platforms has been a hot topic in recent researches. [54] and [5] address the thread mapping strategy in heterogeneous multiprocessor systems by utilizing dynamic thread-to-core assignment. [51] proposes a formal model to characterize threads and cache hierarchy of GPGPUs and generate an optimized thread mapping scheme. [50] also optimizes the shared cache GPGPU, but targets applications with irregular data accesses. [88] and [82] describe novel approaches to schedule threads on DSM systems, taking memory traces and hierarchy into consideration. [9] addresses the problem of application mapping on a Network-on-Chip (NoC) multiprocessor. In [83] and [16], the authors propose an analytical model to characterize programs, machines and costs for multiprocessor platforms with hierarchical memory architectures.

In comparison to these works, our approach differs in two aspects. First, we are optimizing thread mapping on homogeneous many-core platforms with distributed tag directories maintaining cache coherence. As this type of platform is a different processor architecture, the problem of thread-to-core mapping cannot be addressed by the existing methods for CMPs with hierarchical memory system or GPGPUs. Second, we propose the measurement of core distance to quantify the core-to-core transfer latency precisely, rather than using theoretical values as in other work.

3.6 Conclusion

In this chapter, we propose a software approach [58] to optimize the thread-to-core mapping for homogeneous many-core processors with distributed tag directories. By profiling the application and optimizing thread assignments, our approach can reduce the core-to-core transfer latency significantly and improve the application performance by more than 25% over the Linux default strategy.

Chapter 4

Communication-Aware Thread Mapping

In Chapter 3, we define the concept of *core distance*, and optimize thread mapping to reduce on-chip communication overhead on many-core platforms with distrbuted cache tag directories. In this chapter, we target more general host architectures, and propose a thread mapping framework which requires no *a priori* knowledge of the underlying platform. Our framework aims at minimizing the costly inter-chip communication.

4.1 Introduction

Multi-core and many-core platforms have been in wide use during the past few decades. These parallel platforms feature different processor architectures and memory hierarchies. As the most common architecture, a typical hierarchical multiprocessor includes multiple tiles, each consisting of a number of processing cores that share an on-chip Last-Level Cache (LLC). Between separate tiles, they use inter-chip communication for synchronization and data sharing. To take advantage of the faster on-chip cache access, it is intuitive to map threads that share a huge chunk of data to cores on the same processor. In this chapter, we propose a framework to automatically detect processor architectures and communication patterns, and optimize thread-to-core mapping to minimize the costly inter-chip communication latency in SystemC PDES. The key contributions of this chapter are the following:

1. We explore the relationship between communication latency and communication size on hierarchical multiprocessors, and extend our RISC compiler (Section 1.2) to analyze the communication pattern of an application.
2. We adopt the *core distance* concept discussed in Chapter 3 to automatically detect general processor architectures, not restricted to any particular type.
3. We propose a communication-aware thread mapping framework, which automatically identifies communication patterns and processor architectures, and formulates the problem of thread-to-core mapping as graph partitioning. We evaluate our framework with comprehensive experiments.

The rest of the chapter is organized as follows: Section 4.2 reviews related work on communication-based thread mapping optimization. Section 4.3 presents our observations on communication latency with varying communication sizes on a hierarchical multiprocessor. In Section 4.4, we identify three categories of communication patterns in ESL design models, and extend our RISC compiler infrastructure to automatically analyze the communication pattern of a design. Then Section 4.5 proposes our communication-aware thread mapping framework, with the aim of minimizing costly inter-chip communication latency. In Section 4.6, we thoroughly evaluate our framework with benchmarks that follow the three categories of communication patterns. Section 4.7 concludes the chapter.

4.2 Related Work

Thread mapping optimization has been broadly explored in the past few decades, with objectives to minimize the resource contention and communication latency. In [85], the author converts thread mapping to the well-known problem of maximum network flow, and introduces an efficient solution to the two-processor problem, which is derived from the Ford-Fulkerson algorithm. [15, 88, 64] explore various design objectives to improve thread-to-core mapping and achieve better performance. In [4, 19, 13, 46, 59], authors optimize thread mapping based on dynamic information obtained at run time, in order to minimize communication cost. In addition, [21, 22] characterize communication patterns of shared-memory applications by monitoring page table accesses and page faults, and use this information to optimize thread-to-core mapping. In [9], authors propose a compiler approach to improve thread and data mapping on Networks-on-Chip (NoC) multiprocessors. [60] introduces a template library that accepts users' input to reduce costly remote memory accesses on NUMA multiprocessors. [90] proposes the Mix-Scheduling policy to evenly mix threads across cores and achieve thread diversity in every core.

Compared to the previous work, our proposed framework differs in the following aspects:

1. Our framework can automatically detect the processor architecture and generate the communication pattern, which are used as inputs to the graph partitioning tool to optimize thread-to-core mapping. It requires no manual intervention from the user.

2. Our framework is implemented at the user level, which requires no kernel information from the operating system. Thus, our framwork is more flexible and portable than those kernel-level approaches [21, 22].

3. The overhead of our approach is negligible and it can attack the n-processor problem (as opposed to the two-processor problem in [85]).

4.3 Communication Latency on Hierarchical Multiprocessor Platforms

In this section, we first explore the relationship between communication size and communication latency on hierarchical multiprocessors. We implement a SystemC model as Figure 4.1, running on a 2-chip hierarchical multi-core platform. The multiprocessor architecture is depicted in Figure 4.2. Here, each node in Figure 4.1 is implemented as an `SC_MODULE`, initiating an `SC_THREAD` that repetitively receives and sends a data chunk without any processing. All the communications between `SC_MODULE`s go through `sc_fifo` channels. The size of the transferred data chunk is varying from 64 Bytes to 16 MB. In our experiments, we first run the model using our synchronous parallel SystemC simulator with the Linux default scheduling for a few times, and then optimize the thread-to-core mapping in the parallel simulation. Intuitively, it is more efficient to map the two groups of `SC_MODULE`s onto separate chips, so as to avoid the costly inter-chip communication.

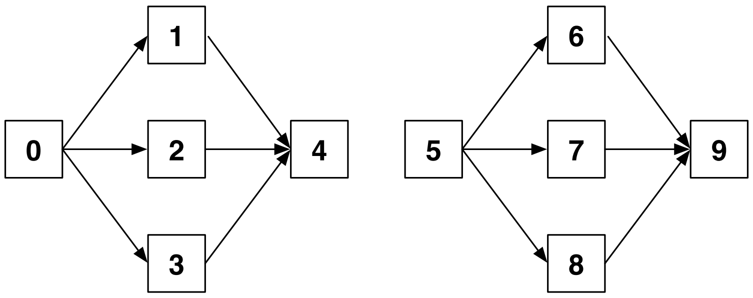


Figure 4.1: Benchmark for communication latency exploration.

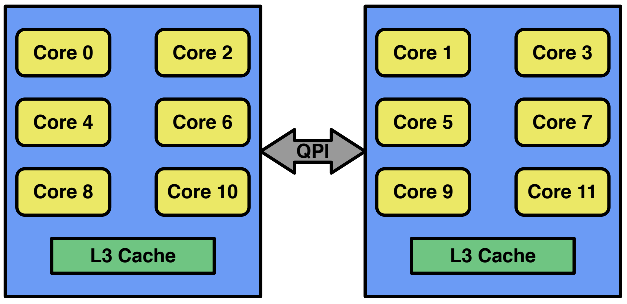
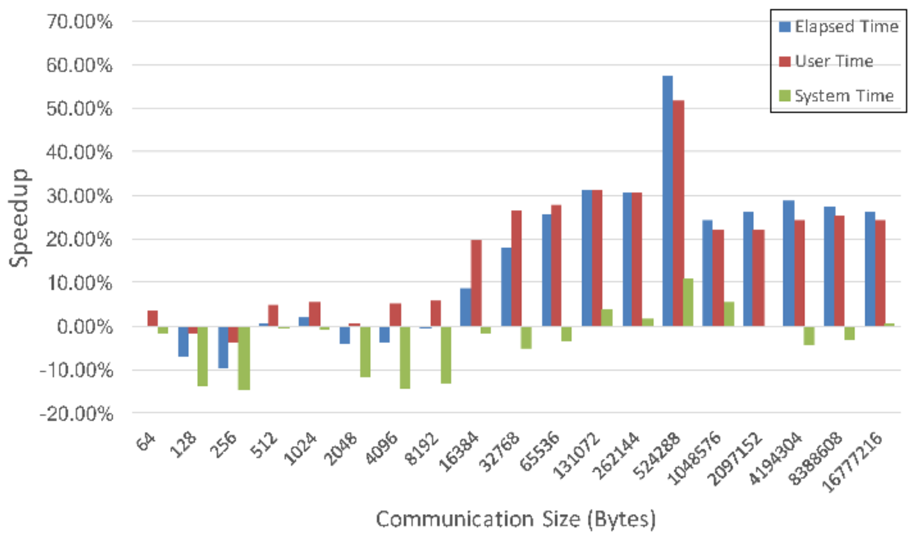
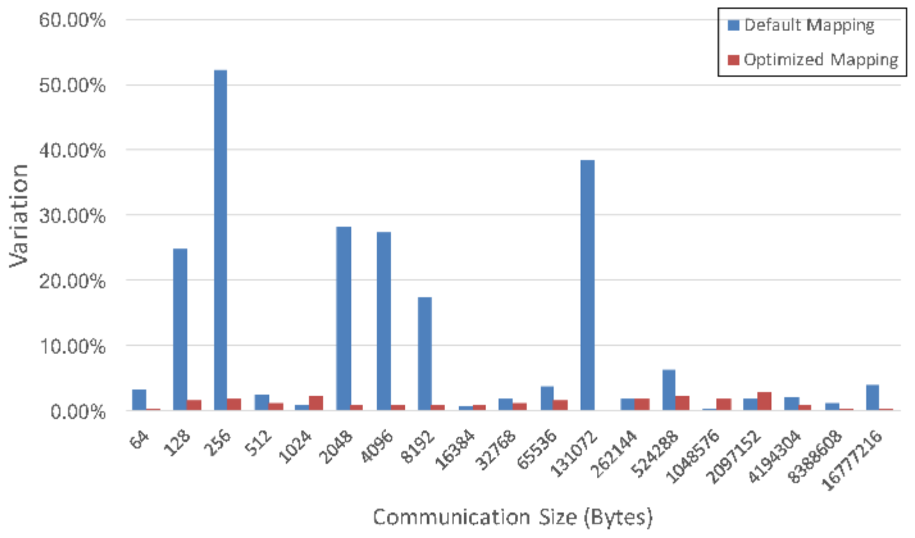


Figure 4.2: A 12-core workstation architecture.



(a) Performance comparison of the Linux default mapping and the optimized mapping.



(b) Performance variation of the Linux default mapping and the optimized mapping.

Figure 4.3: Performance of the Linux default mapping and the optimized mapping.

Figure 4.3a shows the performance gain of the optimized mapping compared to the Linux default mapping. Here, we compare the elapsed time, user time and system time when using the Linux default mapping and the optimized mapping. Figure 4.3b demonstrates the variations of elapsed times with the Linux mapping and the optimized mapping, by comparing the maximum and minimum run time. These two figures allow the following observations:

1. **The optimized mapping performs better when the communication size is large**: In the case that the data chunk is small, e.g., less than 16 KB, the optimized mapping has similar or even worse performance than the Linux default mapping. However, when the communication volume across chips becomes larger, we observe increasing improvement up to 60%. Therefore, it is critical to optimize the thread-to-core mapping when the communication size is large in the model.

2. **The optimized mapping improves only inter-chip communication**: When the data chunk becomes larger than 512 KB, the performance gain of the optimized mapping flattens as not all the shared data can fit into the Last-Level Cache (LLC). In this case, part of shared data stays in the main memory, and our optimized thread mapping only mitigates the costly inter-chip communication.

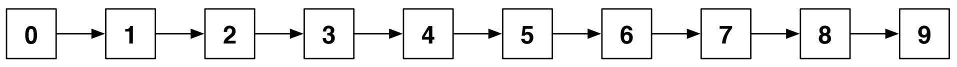
3. **The optimized mapping mainly affects user time**: While the system time varies little between the Linux default mapping and the optimized mapping, the speedup of user time is quite similar to that of the total elapsed time. This matches our point in Section 1.5 that it is difficult to improve system time by user-level thread mapping.

4. **The performance of the optimized mapping is consistent**: Compared to the Linux default mapping, our optimized mapping yields much more stable performance. While the run time with the Linux default mapping varies by more than 50%, the run times with our optimized mapping stay quite close.

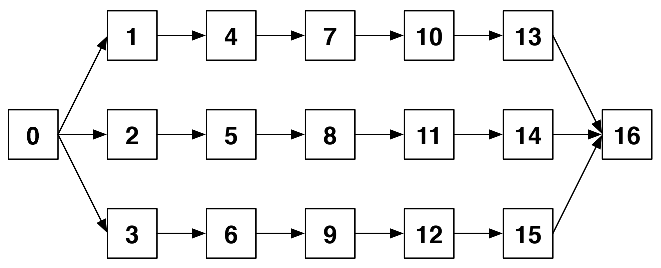
Based on these observations, we conclude that only the large communication size matters in thread mapping optimization. On the 12-core multiprocessor platform as Figure 4.2, the optimized mapping achieves little speedup for shared data that is smaller than 16 KB. Thus, we propose the *Scaling Communication Pattern* to simplify the communication graph of a design model, which eliminates the lightweight communications (e.g., whose communication size is smaller than the scaling parameter, say 16 KB for the host architecture as Figure 4.2).

4.4 Communication Patterns of System Level Designs

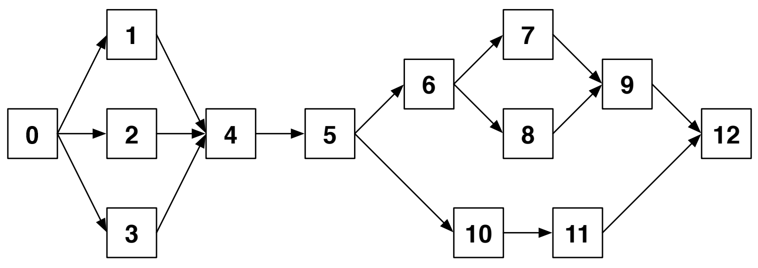
In system level designs, inter-thread communication is usually wrapped in channels. Thus it is possible to statically analyze the communication pattern of a system level design, by counting the size of the shared data chunk in the channel. In [76], we present an approach to automatically extract thread communication graphs from SystemC source code. By extending our dedicated RISC compiler (Section 1.2) with the approach in [76] and calculating the communication size of every channel in AST, we can generate the *communication pattern* of an ESL design model.



(a) Pipeline pattern.



(b) Parallel pattern.



(c) General pattern.

Figure 4.4: Different communication patterns in ESL design models.

Figure 4.4 lists three common types of communication patterns in system level designs. Here, each node of the communication pattern is a thread *segment*, and an edge represents the communication between two thread segments. Both nodes and edges have weights, which are computation loads and communication sizes respectively. In Figure 4.4a, except the source and drain, every thread segment receives a data chunk from its predecessor and sends another to its successor. All thread segments operate in a pipelined fashion. In Figure 4.4b, the source segment distributes data chunks to multiple parallel thread segments, and the drain collects results from the parallel pipelines. The communication pattern in Figure 4.4c is most general, which is a combination of the previous two patterns.

Even though we do not have complete numbers for all the existing system level designs, we count the communication patterns of the ESL design models used in our lab for the past 15 years. Table 4.1 lists the numbers of different communication patterns used in our designs, including our SpecC and SystemC implementations, and the S2CBenchmark models.

Table 4.1: Numbers of different communication patterns in our ESL design models.

Communication Pattern	Number	Percentage
Pipeline pattern	26	44%
Parallel pattern	17	28%
General pattern	17	28%
Total	60	100%

In Table 4.1, it is clear that all the communication patterns of our design models are covered by the three types. Some examples of the pipeline pattern include the monochrome JPEG encoder, Data Encryption Standard (DES) cipher and all the S2CBenchmark models. Other applications, e.g., the Mandelbrot set renderer, H264 decoder, and polychrome JPEG decoder, follow the parallel communication pattern. As for the general pattern, it is used in the implementations of the Canny edge detector, Networks-on-Chip (NoC) particle simulator and DVD player.

4.5 Communication-Aware Thread Mapping Framework

In Chapter 3, we introduced *core distance* to quantify core-to-core communication latency on multi-core and many-core platforms. Combined with Operating System (OS) query or user input on the clustering of core IDs, core distance profiling can be used to automatically detect the underlying processor architecture.

Algorithm 7 Processor Architecture Detection Algorithm

```
Input:
    $coredist_{i,j}$ where $i, j \in ProcCores$ and $i \neq j$
    $\forall ProcCores_k$ where $ProcCores_k$ is the set of all the core IDs on Processor $k$
 1: $MaxVaration \leftarrow 0$
 2: for all $ProcCores_k$ do
 3:     calculate the $varation_k$ of $coredist_{i,j}$ on Processor $k$
 4:     if $varation_k > MaxVaration$ then
 5:         $MaxVaration \leftarrow varation_k$
 6:     end if
 7: end for
 8: if $MaxVaration \leq threshold$ then
 9:     return Hierarchical Multiprocessor Architecture
10: else
11:     return Ring-Based Many-Core Architecture with Distributed Tag Directories
12: end if
```

Algorithm 7 lists the pseudocode of our algorithm to detect the overall processor architecture. Here, the input $coredist_{i,j}$ can be measured by using our *pingpong* function presented in Section 3.2, and $ProcCores_k$ lists all the core IDs on Processor k, which can be specified by the user or rely on OS query (e.g., parsing the system file `/proc/cpuinfo`). Then, it calculates the varition of *core distance* between cores on the same processor. Based on our observations of core distance on multi-core and many-core platforms in Section 3.2, we see that if the core distances on the same processor stays close, it is a hierarchical multiprocessor. On the other hand, if the core distances match the blue line in Figure 3.5, it is a ring-network many-core processor with distributed tag directories. The decision criteria in Line 8 can be extended to take more types of processor architectures into consideration. Here we only distinguish between the two common types of platforms discussed in Section 3.2, i.e., hierarchical multi-core platforms and ring-based many-core platforms with distributed tag directories. Empirically, the *threshold* in Algorithm 7 can be 125%.

With the communication pattern and processor architecture available, we can effectively improve the thread-to-core mapping in SystemC PDES. Here, our goal is to minimize the costly inter-chip communication latency[1]. Figure 4.5 depicts our *communication-aware* thread mapping framework. It consists of three parts, i.e., the static analysis in the compilation phase, the dynamic profiling in the pre-run phase, and the online scheduling in the simulation. The major functional blocks of our framework are as follows:

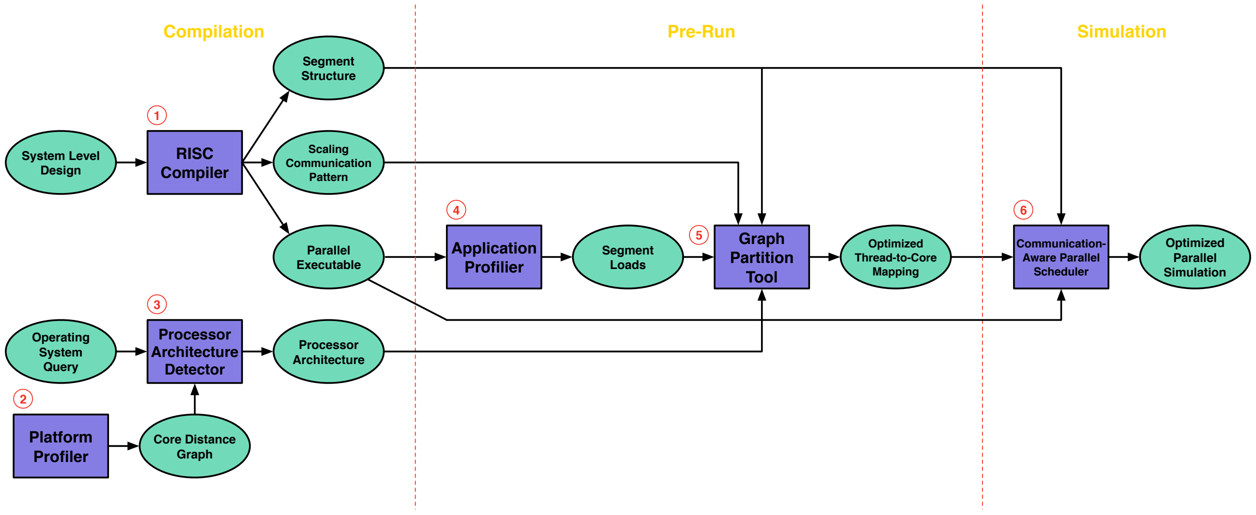


Figure 4.5: Communication-aware thread mapping framework.

1. **RISC Compiler**: In the compilation phase, we first utilize our dedicated RISC compiler [56] to statically analyze the design model, and automatically generate the segment structure and scaling communication pattern, as presented in Section 2.4.3 and 4.4. Also, the RISC compiler will generate the parallel executable, which is linked against our PDES library.

[1]Chapter 3 presents a novel technique to reduce the expensive on-chip communication on many-core platforms.

2. **Platform Profiler**: Second, the platform profiler generates the core distance graph of the underlying platform by using the proposed *pingpong* function from Section 3.2.

3. **Processor Architecture Detector**: The third step in the compilation phase is to detect the processor architecture based on the OS query and core distance graph. Algorithm 7 describes this procedure.

4. **Application Profiler**: Next, our framework dynamically profiles the parallel executable in the pre-run phase. As discussed in Section 2.4.4, threads in system level designs are usually implemented in a loop structure. Here, we run the parallel executable for a few iterations, in order to measure the workload of each thread segment. Typically, the workload of a segment varies only little in different iterations.

5. **Graph Partition Tool**: When the segment structure, communication pattern, segment loads and processor architecture are ready, we optimize the thread-to-core mapping in the parallel SystemC simulation. For the common hierarchical multiprocessors, on-chip communication latencies between cores on the same processor are identical, as these cores share the Last-Level Cache (LLC). However, the inter-chip communication is much more expensive than on-chip communication, as shown in Figure 3.4. In order to improve the communication overhead in the parallel simulation, it is necessary to minimize the costly inter-chip communication. Besides, the workloads on separate processors must be balanced in order to utilize the processing units efficiently. Thus, our optimization objectives are twofold:

   - Minimize inter-chip communication volume;
   - Balance workloads on separate processors.

   This is a classic graph partitioning problem, to which the optimal solution is proven to be NP-complete [2]. Thus, we utilize heuristic algorithms to efficiently generate the desired optimized thread-to-core mapping. Here, we are using the open-source package

METIS [47], which implements the multilevel Recursive-Bisection (RB) and multilevel k-way (K-way) algorithms, to solve the graph partitioning problem. Our framework converts the communication pattern and segment loads into the format of the METIS graph file[2], and specifies the number of target partitions as the number of separate processors. Then, the METIS library takes these inputs and efficiently generates an optimized thread-to-core mapping, which partitions threads for different processors.

6. **Communication-Aware Parallel Scheduler**: The last step of our framework is to follow the optimized thread-to-core mapping at run time. Our communication-aware scheduler will dispatch thread segments to the target processors, based on the partitioning results from METIS.

Table 4.2: Comparison of the Linux scheduling and our thread mapping framework.

	Linux Scheduling	Our Thread Mapping Framework
Segment Structure	Unaware of segment structure. Able to identify whether threads are I/O bound or computation bound. Favor interactive threads.	Aware of segment structure. Make better prediction of workload based on the static segment information.
Communication Pattern	Unaware of communication pattern. Randomly distribute threads over separate processors.	Aware of communication pattern. Optimize thread mapping to minimize inter-chip communication.
Processor Architecture	Aware of processor architecture. Rebalance workload on different processing cores in an ad-hoc fashion.	Profile processor architecture and make fast adjustments by using workload information at run time.

Table 4.2 compares the differences between the Linux default scheduling and our thread mapping framework. As discussed in Section 2.4.4, Linux is unaware of the segment structure of a design model. It only identifies whether a thread is I/O bound (e.g., interactive processes) or computation bound (e.g., batch processes), and favors interactive processes. In

[2]Specifically, the computation load of a thread segment is defined as the node weight, and the communication size between threads is defined as the edge weight.

comparison, by using the segment information from the RISC compiler, our segment-aware profiler makes better prediction of the run time of a thread segment. Also, Linux is unaware of the communication pattern of the design, and thus it randomly distributes threads to separate processors and may trigger more costly inter-chip communication. To fix this issue, our graph partition tool takes the communication pattern and processor architecture as inputs, and generates an optimized thread mapping that minimizes communication volume across processors. Regarding the processor architecture, while Linux has full access to the kernel information and rebalances workloads in an ad-hoc fashion, our framework profiles the processor architecture and makes fast adjustments at the user level when the thread workload changes.

4.6 Experimental Evaluation

In this section, we evaluate our framework with communication-intensive benchmarks to demonstrate its performance gains. The whole set of benchmarks are generated by Task Graphs for Free (TGFF) [20], and evaluated on two hierarchical multiprocessors. Table 4.3 lists their hardware specifications and Figure 4.2 and 4.6 depict their architectures.

Table 4.3: Workstations used for evaluation.

Host	**12-Core**	**16-Core**
Processor	Intel Xeon X5650	Intel Xeon E5-2680
CPU frequency	2.67 GHz	2.7 GHz
Physical CPUs	2	2
Cores/CPU	6	8
Hyperthreading	off	off
Total HW threads	12	16

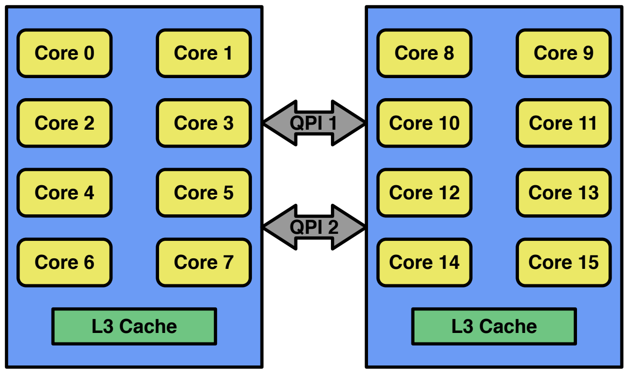


Figure 4.6: A 16-core workstation architecture.

4.6.1 Task Graph Benchmarks

In Section 2.5.1, we extended TGFF to output actual SystemC models. Here, we use the same tool to generate SystemC benchmarks which follow the three categories of communication patterns. Specifically, each `SC_THREAD` has a single segment, and repetitively transfers data chunks through `sc_fifo` channels without any processing. Thus, the thread segment in this evaluation has little computation load but intensive communication overhead.

Next, we demonstrate the evaluation results for the three categories of communication patterns separately.

4.6.1.1 Pipeline Communication Patterns

For the first experiment, we evaluate our framework with SystemC models which follow the pipeline communication pattern like Figure 4.4a. Here, the sizes of the data chunks in `sc_fifo` channels are generated by TGFF as attributes, randomly distributed in the range from 500 KB to 1.5 MB. Also, we generate three sets of task graphs which have different amount of parallelism. The average number of parallel threads per core is chosen in the range of less than 1, 1 to 2, or 3 to 5. Each set of task graphs consists of 30 benchmarks, running on both workstations.

Table 4.4: Performance of different graph partitioning algorithms for pipeline communication patterns (Figure 4.4a).

Par	12-Core Host					16-Core Host				
	Linux Runtime	K-way Speedup	K-way Overhead	RB Speedup	RB Overhead	Linux Runtime	K-way Speedup	K-way Overhead	RB Speedup	RB Overhead
less than 1 thread per core	*56.76s*	**+28.55%**	*0.001s*	+23.66%	*0.001s*	*39.97s*	**+19.67%**	*0.001s*	+19.14%	*0.001s*
1 to 2 threads per core	*108.74s*	**+18.90%**	*0.001s*	+15.06%	*0.001s*	*77.23s*	**+11.59%**	*0.001s*	+7.99%	*0.001s*
3 to 5 threads per core	*243.07s*	**+8.83%**	*0.001s*	+7.01%	*0.001s*	*160.81s*	**+3.92%**	*0.001s*	+3.50%	*0.001s*

Table 4.4 shows the average performance gains of our thread mapping framework over the 30 benchmarks, deploying two different graph partitioning algorithms (*K-way* and *RB*) in METIS. We use the Linux default scheduling as the baseline of our evaluation. The first column *Par* in the table refers to the average number of parallel threads per core in the SystemC models. Next, the table lists the average performance of the parallel simulation, using different thread mapping schemes, together with the overhead of our optimization. Here, the simulation time of *K-way* and *RB* already includes the overhead of our framework.

Table 4.4 allows the following observations:

1. **Our framework performs significantly better than Linux with negligible overhead**: Our framework yields much better performance than Linux for all the three sets of benchmarks, using either graph partitioning algorithm. While the maximum speedup reaches 28%, the overhead of our optimization is negligible compared to the total runtime.

2. **Our framework shows better performance when most of the shared data can fit into LLC**: With an increasing number of parallel threads in the benchmarks, the performance speedup of our optimization decreases. As discussed in Section 4.3, our optimized thread-to-core mapping only improves inter-chip communication. When

there are more parallel threads running in the simulation, not all the shared data can fit into the Last-Level Cache (LLC). Currently, our framework can not optimize main memory access. This will be one direction of our future work.

Figure 4.7 and 4.8 shows the performance of our framework using different graph partitioning algorithms, for each individual benchmark on both hosts. Here, we make another two observations:

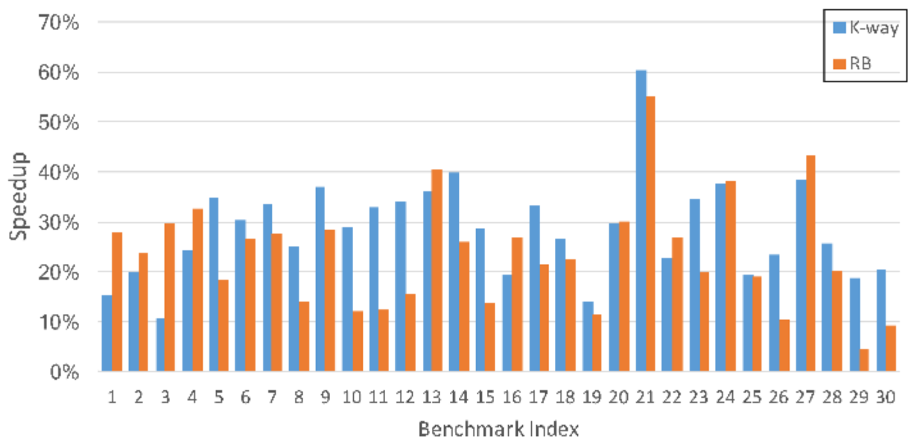
3. **The graph partitioning results affect the simulation performance**: In Figure 4.7 and 4.8, the *K-way* partitioning algorithm yields better performance than *RB* for some benchmarks, while it is opposite for others. Clearly, a better graph partitioning result leads to higher performance gain. While both partitioning algorithms (*K-way* and *RB*) are only heuristics, they generate significantly better thread mapping than Linux.

4. **Our framework shows better performance than Linux with only few exceptions**: Even though the 180 benchmarks used in our experiment have different amounts of thread parallelism and communication size, our framework performs better than the Linux default scheduling for almost all the benchmarks. Only in some rare cases when Linux is "lucky" to find an optimal mapping, the graph partitioning results are slightly worse.

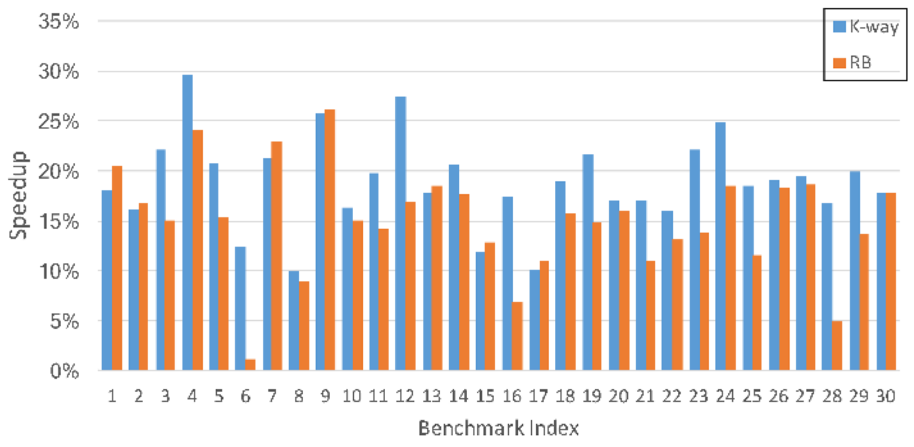
4.6.1.2 Parallel Communication Patterns

Next, we use the extended TGFF to generate SystemC benchmarks with parallel communication patterns as Figure 4.4b. Table 4.5 shows the experimental results.

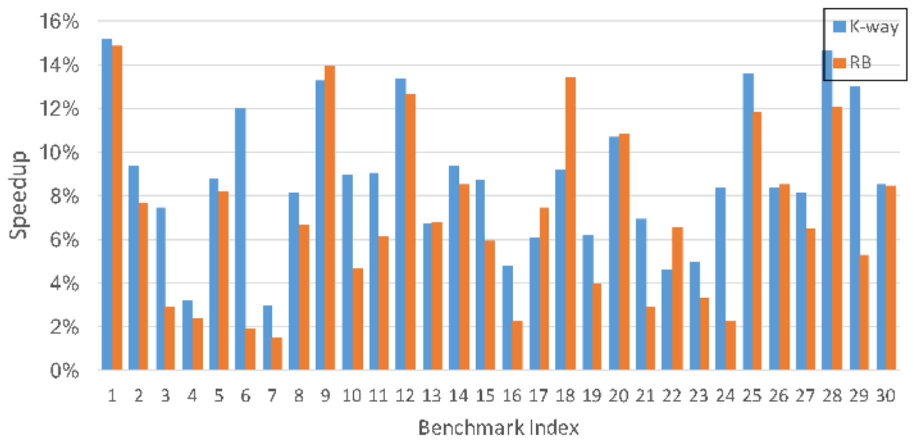
In addition to the same observations from Table 4.4, Table 4.5 allows the following observation:



(a) Less than 1 thread per core.

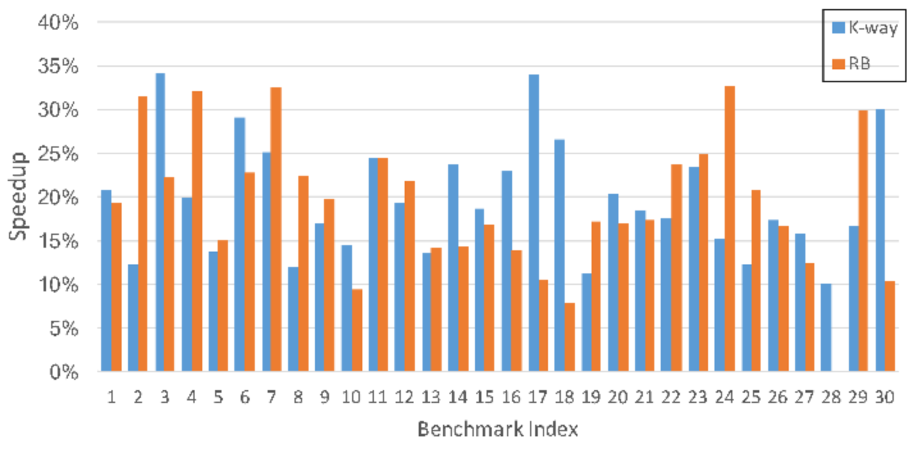


(b) 1 to 2 threads per core.

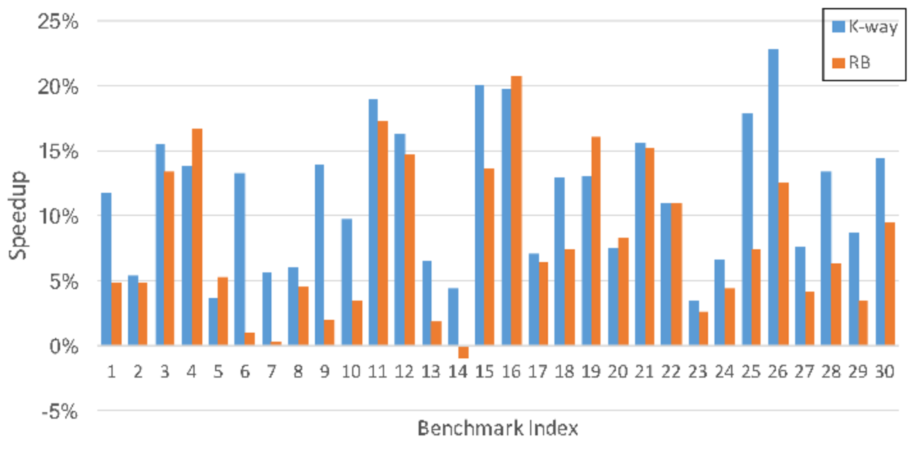


(c) 3 to 5 threads per core.

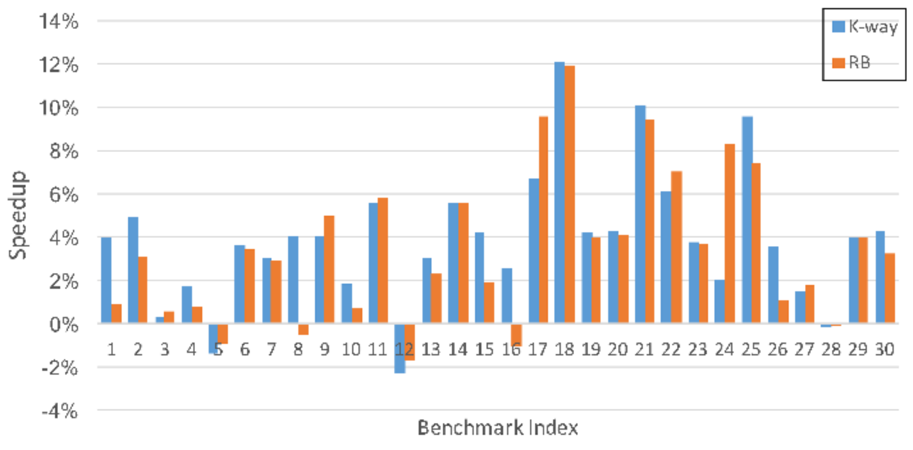
Figure 4.7: Performance comparison for pipeline communication patterns (Figure 4.4a) on a 12-core host.



(a) Less than 1 thread per core.



(b) 1 to 2 threads per core.



(c) 3 to 5 threads per core.

Figure 4.8: Performance comparison for pipeline communication patterns (Figure 4.4a) on a 16-core host.

Table 4.5: Performance of different graph partitioning algorithms for parallel communication patterns (Figure 4.4b).

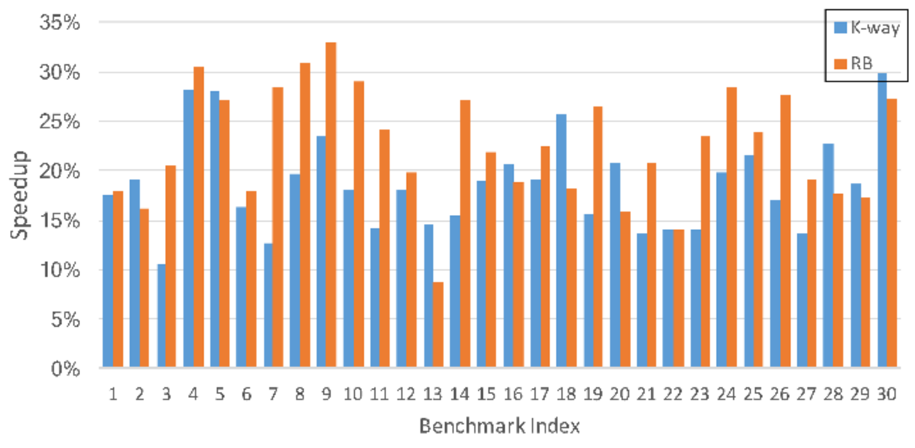
Par	12-Core Host					16-Core Host				
	Linux Runtime	K-way Speedup	K-way Overhead	RB Speedup	RB Overhead	Linux Runtime	K-way Speedup	K-way Overhead	RB Speedup	RB Overhead
less than 1 thread per core	*67.33s*	+18.71%	*0.001s*	**+22.48%**	*0.001s*	*56.23s*	+19.50%	*0.001s*	**+20.93%**	*0.001s*
1 to 2 threads per core	*127.75s*	**+19.00%**	*0.001s*	+18.71%	*0.001s*	*111.91s*	+11.05%	*0.001s*	**+11.40%**	*0.001s*
3 to 5 threads per core	*273.71s*	**+10.21%**	*0.001s*	+9.66%	*0.001s*	*188.70s*	**+4.98%**	*0.001s*	+4.87%	*0.001s*

5. **The graph partitioning algorithms generate better results for simpler patterns**: Compared to Table 4.4, we observe that the performance gain of our framework for parallel communication patterns is slightly worse than that for pipeline communication patterns, in the case that the average number of parallel threads per core is less than 1. Clearly, the *K-way* and *RB* partitioning algorithms yield better results for the simple pipeline patterns, achieving a speedup of 28%, while it is only 22% for the parallel communication patterns.

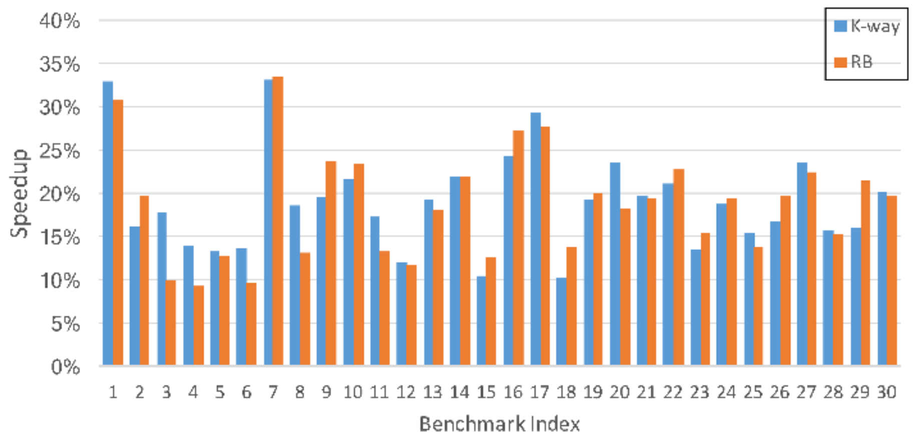
Figure 4.9 and 4.10 shows the performance gain of our optimization for each individual benchmark with the parallel communication pattern on the 12-core and 16-core hosts. Again, our framework performs much better than the Linux default scheduling for the large majority of benchmarks.

4.6.1.3 General Communication Patterns

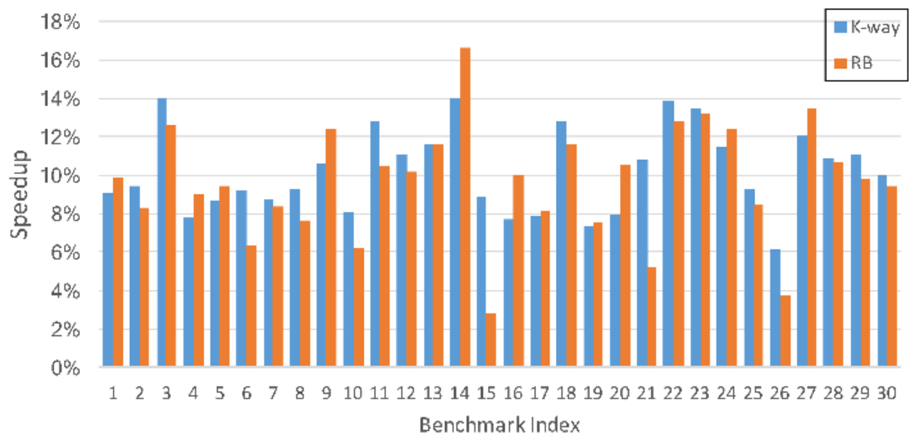
Finally we evaluate our optimization with benchmarks that follow the general communication patterns, like Figure 4.4c. Table 4.6 shows the average performance gain of our framework on both hosts. Figure 4.11 and 4.12 demonstrate the performance for each individual benchmark. Here, we make another observation:



(a) Less than 1 thread per core.

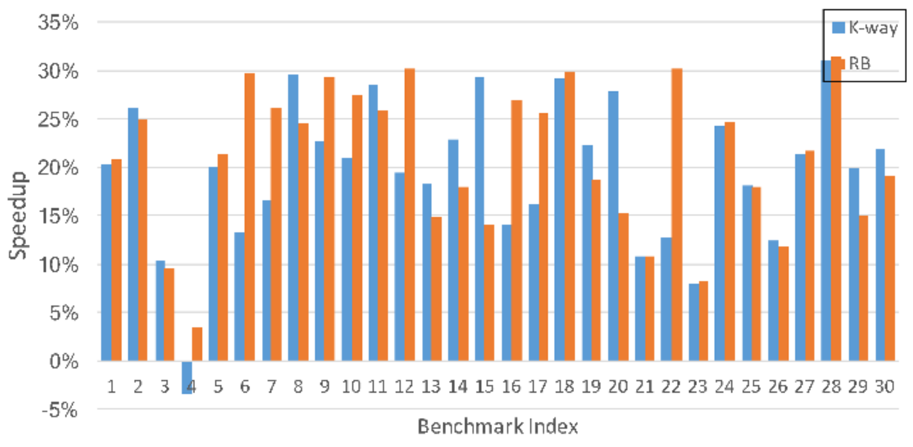


(b) 1 to 2 threads per core.

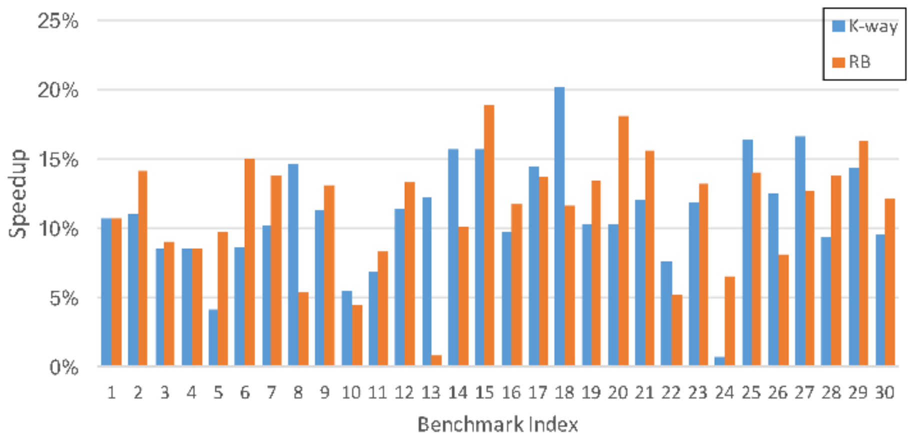


(c) 3 to 5 threads per core.

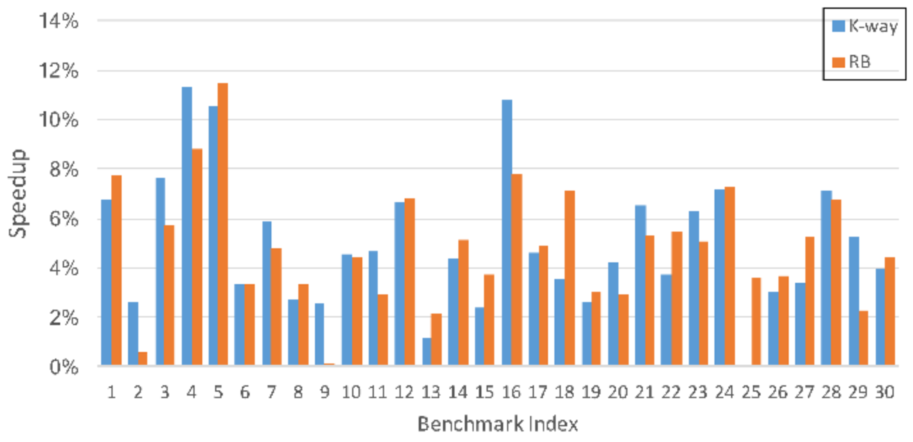
Figure 4.9: Performance comparison for parallel communication patterns (Figure 4.4b) on a 12-core host.



(a) Less than 1 thread per core.



(b) 1 to 2 threads per core.



(c) 3 to 5 threads per core.

Figure 4.10: Performance comparison for parallel communication patterns (Figure 4.4b) on a 16-core host.

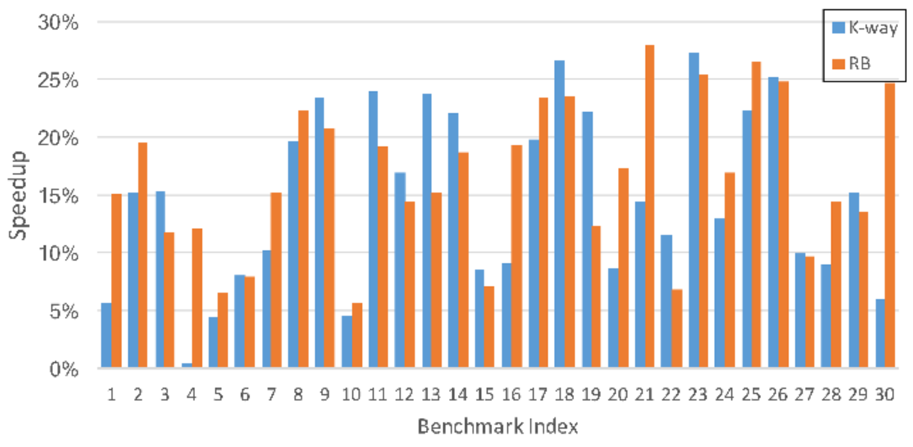
Table 4.6: Performance of different graph partitioning algorithms for general communication patterns (Figure 4.4c).

Par	12-Core Host					16-Core Host				
	Linux Runtime	K-way Speedup	K-way Overhead	RB Speedup	RB Overhead	Linux Runtime	K-way Speedup	K-way Overhead	RB Speedup	RB Overhead
less than 1 thread per core	*134.30s*	+14.80%	*0.001s*	**+16.66%**	*0.001s*	*116.35s*	+16.59%	*0.001s*	**+20.01%**	*0.001s*
1 to 2 threads per core	*252.90s*	**+13.77%**	*0.001s*	+11.67%	*0.001s*	*205.15s*	+14.39%	*0.001s*	**+15.92%**	*0.001s*
3 to 5 threads per core	*769.86s*	**+14.41%**	*0.001s*	+14.28%	*0.001s*	*732.51s*	+12.88%	*0.001s*	**+13.35%**	*0.001s*

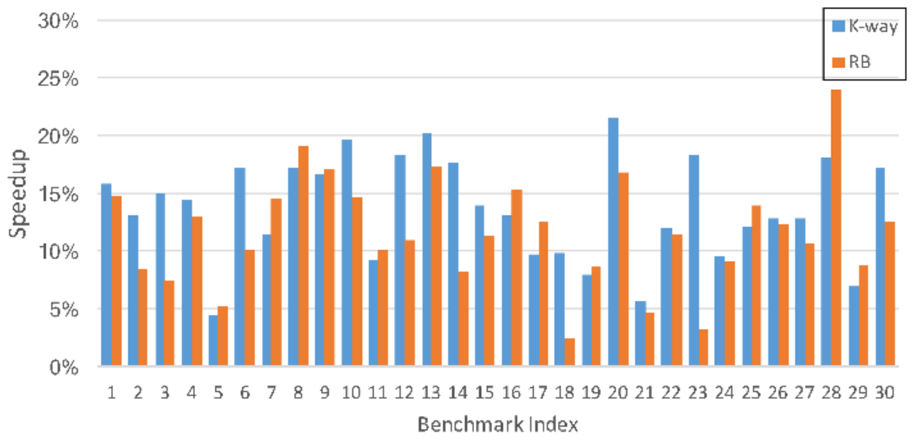
6. **The performance gain of our framework is more stable for general communication patterns**: Compared to the experiments with pipeline and parallel communication patterns, our framework achieves a more stable speedup for benchmarks with general communication patterns. While the performance gain of our framework is up to 16% on the 12-core host in cases of less than 1 thread per core, it remains 14% when the parallelism increases to 3 to 5 threads per core. In the general communication pattern, each thread segment has more dependencies on others, and thus there are less threads and shared data chunks active simultaneously.

4.7 Conclusion

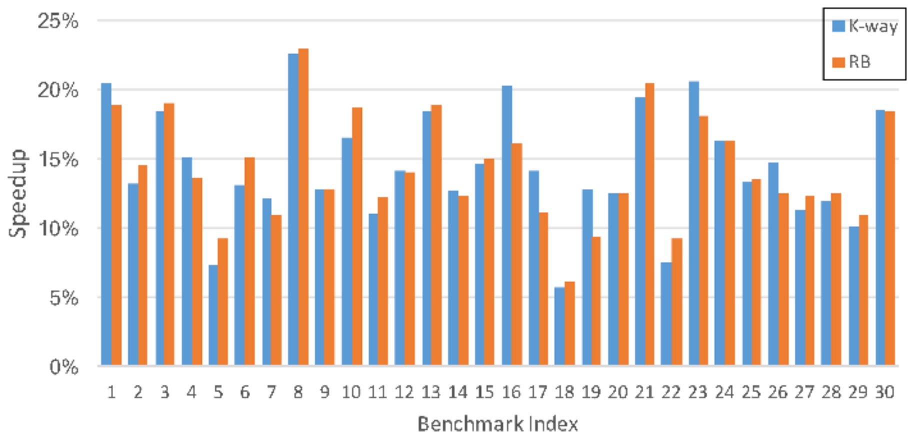
In this chapter, we proposed a communication-aware thread mapping framework, which automically analyzes communication patterns and detects processor architectures. By using an open-source graph partitioning tool and our RISC infrastructure, our framework generates an optimized thread-to-core mapping to minimize costly inter-chip communication. In our comprehensive evaluation, our framework consistently shows a significant speedup compared to the Linux default scheduling. Also, we observe negligible overhead in our thread mapping framework.



(a) Less than 1 thread per core.

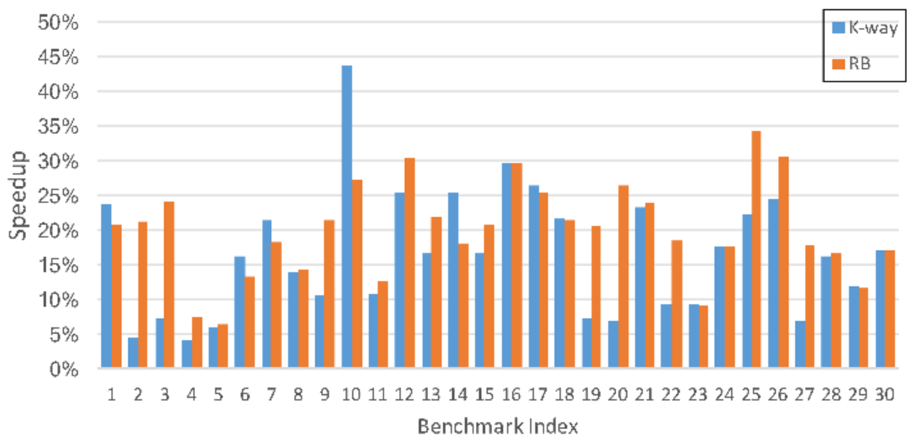


(b) 1 to 2 threads per core.

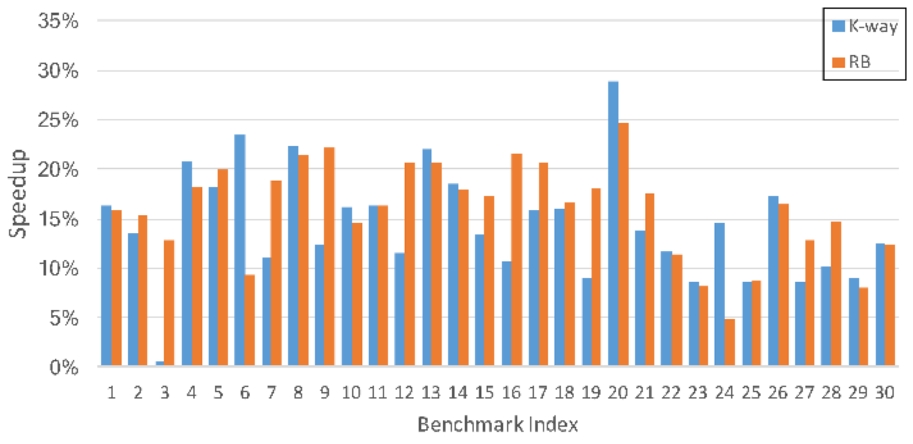


(c) 3 to 5 threads per core.

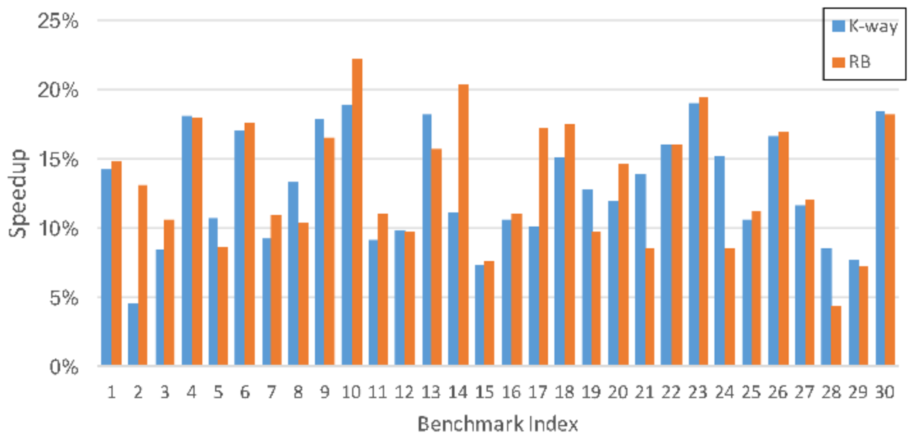
Figure 4.11: Performance comparison for general communication patterns (Figure 4.4c) on a 12-core host.



(a) Less than 1 thread per core.



(b) 1 to 2 threads per core.



(c) 3 to 5 threads per core.

Figure 4.12: Performance comparison for general communication patterns (Figure 4.4c) on a 16-core host.

Chapter 5

Computation- and Communication-Aware Thread Mapping

In Chapter 2, we proposed a segment-aware multi-core scheduler, addressing the load balancing problem in the parallel SystemC simulation. Then in Chapter 4, we introduced a communication-aware thread mapping framework, which minimizes costly inter-chip communication. However, a real-world application may consist of both significant computation and communication load, for which our previous two methods may not yield a satisfactory performance. In this chapter, we integrate the previous two methods and evaluate the integration with a real-world graphics application.

5.1 Integration of Our Methods

Before integrating our proposed methods in Chapter 2 and 4, we first present the computation weight of a real-world application. Then we propose our integration which optimizes thread mapping for reducing both computation and communication time in ESL simulation.

5.1.1 Computation Weight in Real-World Examples

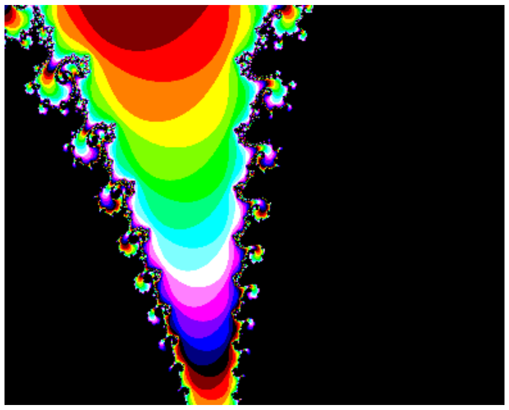
In the Mandelbrot set renderer example [61], an iterative calculation is performed for each pixel in the complex plot area, and it breaks the loop when the escape conditions meet. The designer can determine the maximum depth of this iterative calculation. The higher the maximum depth, the more details and accuracy appear in the result image. Figure 5.1 shows images of the same Mandelbrot set with different calculation depths. Here, we use 16 different colors[1] to show the number of executed iterations for each pixel. Clearly, the Mandelbrot set image with a maximum calculaton depth of 1024 is much more detailed than images with lower calculation depths.

Figure 5.2 shows the computation weight in the Mandelbrot set renderer example, when running on the 12-core host as depicted in Figure 4.2. All the Mandelbrot set renderer examples have the same plot area, only differing in the maximum depth of calculation for each pixel. When the maximum calculation depth is 4, a lot of details are missing in the Mandelbrot set image (Figure 5.1a), and computation takes up 57% of the total execution time. This weight climbs up quickly when the calculation depth increases, reaching 99% for a calculation depth of 1024. Clearly, computation is more intensive than communication in the Mandelbrot set renderer example on the hierarchical multiprocessor, which is also true for many real-world examples. The communication-aware thread mapping framework

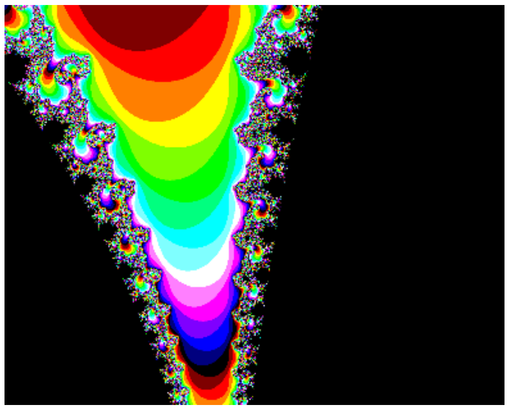
[1] 0: black, 1: brown, 2: red, 3: orange, 4: yellow, 5: light green, 6: green, 7: blue green, 8: turquoise, 9: light blue, 10: white, 11: pink, 12: light pink, 13: purple, 14: blue, 15: dark blue.

(a) Calculation depth: 4

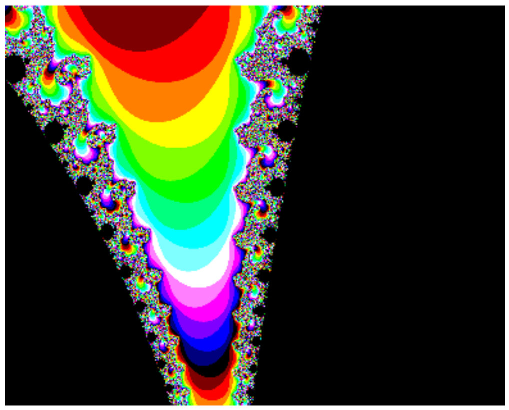
(b) Calculation depth: 16



(c) Calculation depth: 64



(d) Calculation depth: 256



(e) Calculation depth: 1024

Figure 5.1: Mandelbrot set images with different calculation depths.

proposed in Chapter 4 may yield little speedup for this category of examples. Thus, it is important to determine the correct thread mapping optimization for an example.

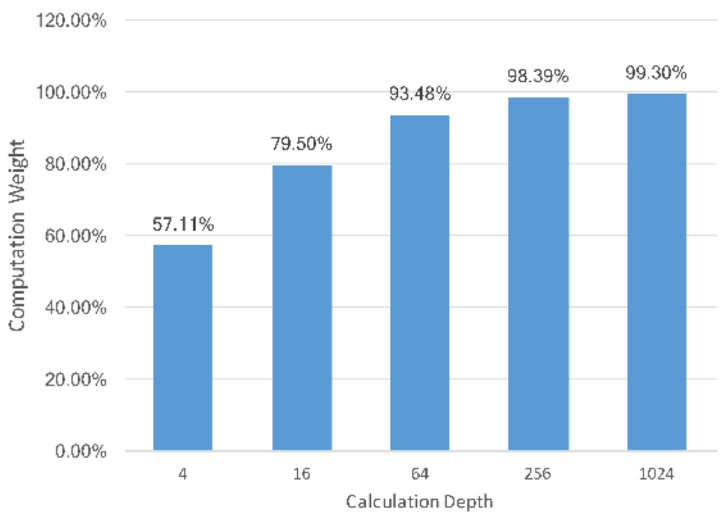


Figure 5.2: Computation weight of a Mandelbrot set renderer example.

5.1.2 Computation- and Communication-Aware Scheduling Algorithm

In order to determine the thread mapping optimization suitable for an example, it is intuitive to first identify the type of the application through profiling. If the example is mainly computation-intensive, we should apply our segment-aware mutli-core scheduler from Chapter 2. On the other hand, if the inter-thread communication takes up the majority of the simulation time, the communication-aware thread mapping framework from Chapter 4 can guarantee a significant speedup compared to the Linux default scheduling. For examples and benchmarks that have both intensive computation and communication, we propose an integration of the previous two methods.

In Chapter 2, the segment-aware multi-core scheduler aims at utilizing the processing units as efficiently as possible. It is possible that one thread is migrated between cores or even processors for the sake of computation efficiency, which creates redundent inter-chip communication. On the other hand, the objectives of our communication-aware thread mapping in Chapter 4 are inter-chip communication minimization and workload balancing on separate

Algorithm 8 Computation- and Communication-Aware Scheduling Algorithm

```
Input:
    Current thread $th_{curr}$
    Next segment $SegID_{next}$
 1: $th_{curr}.T_{end} \leftarrow$ CurrentCycles()
 2: $RunTime[th_{curr}.SegID] \leftarrow th_{curr}.T_{end} - th_{curr}.T_{start}$
 3: $th_{curr}.SegID \leftarrow SegID_{next}$
 4: while true do
 5:   while READY $\neq \varnothing$ do
 6:     if $\exists$ READY$_i \neq \varnothing$ and ThreadNum($Proc_i$) $< MaxThreadsPerProc$ then
 7:       $th_{next} \leftarrow$ pop(READY$_i$)
 8:       $th_{next}.T_{start} \leftarrow$ CurrentCycles()
 9:       dispatch($th_{next}$, $Proc_i$)
10:     else
11:       suspend($th_{curr}$)
12:     end if
13:   end while
14:   Delta cycle $\delta \leftarrow \delta + 1$
15:   process any requested updates in primitive channels
16:   process any delta notifications
17:   if READY $= \varnothing$ then
18:     advance simulation time
19:     process any timed notifications
20:     if READY $= \varnothing$ then
21:       terminate the simulation
22:     end if
23:   end if
24:   sort threads $\forall th \in$ READY$_i$ in decreasing order of $RunTime[th.SegID]$
25: end while
```

processors. However, the thread dispatch order on every processor is determined by Linux, and simulation may run longer due to the inferior dispatch order.

Algorithm 8 lists the pseudocode of our integrated *computation- and communication-aware* scheduling algorithm, which makes a trade-off between the two methods in Chapter 2 and 4. While it reduces inter-chip communication as our framework in Chapter 4, it dispatches ready threads on every processor following the segment-aware LJF policy.

Compared to Algorithm 2 in Section 2.4.4, Algorithm 8 has the following differences:

1. **The READY queue is divided into a set of queues for different processors**: In our integrated algorithm, the READY queue consists of a set of queues, each containing ready-to-run threads for a specific processor.

   $$READY = \cup READY_i$$

   Here, READY_i contains all the ready threads mapped to $Processor_i$, in the order of LJF. The thread-to-processor mapping is determined by our communication-aware optimization in Chapter 4, and a thread will only be dispatched to its target processor. Then, Linux determines when and on which core of the target processor this thread will run. As Linux has access to run-time processor information, it usually makes better decisions than manually fixing the thread affinity to a single core in Algorithm 2.

2. **The number of threads running on a processor is greater than the number of cores**: In Algorithm 8, when there exists a thread in READY_i and the number of running threads on $Processor_i$ is less than $MaxThreadsPerProc$, the scheduler will dispatch the first thread in READY_i to $Processor_i$. As the threads in the application have high communication load and need to halt for data fetching and recording, $MaxThreadsPerProc$ is usually set to be greater than the number of processing cores on the processor. Empirically, $MaxThreadsPerProc$ is defined as 2 times the number of cores on a processor, to achieve the best run-time performance.

Also, our thread mapping framework (Figure 4.5) needs fine-tuning for the integration. Here are a few changes:

1. **The graph partition tool needs to be invoked dynamically in the simulation**: In the parallel simulation, as not all the threads are active simultaneously, the static communication pattern only guarantees that the total workloads on distinct processors are balanced. However, in any delta cycle, the static thread-to-core mapping from Fig-

ure 4.5 may distribute unbalanced workloads to separate processors[2]. In order to cope with the problem of static thread mapping, the scheduler invokes the METIS library for any dynamic communication pattern, and then memorizes the METIS partition result as the same pattern may appear again. In the dynamic communication pattern, the computation loads of all the inactive threads are set to zero, and all the communication sizes keep the same. Next, the dynamic patition result from METIS determines the READY queue (READY_i) that a thread is pushed into.

2. **The communication-aware scheduler dispatches threads according to LJF**: Rather than using the Linux default dispatching policy, the communication-aware parallel scheduler applies Algorithm 8 to dispatch ready threads. As identified in Chapter 2, Longest Job First (LJF) is a better than default thread dispatch policy, when the scheduler has accurate predicition of thread run times.

In our integration, the dynamic partition may incur more inter-chip communication than the static one from Chapter 4, and some processing cores may stay idle while there exist ready threads for other processors. These are the trade-offs we make between the previous two methods, so as to efficiently utilize processing units and minimize inter-chip communication at the same time.

5.2 Experimental Evaluation

In this section, we evaluate our integration with the pipelined Canny edge detector introduced in Chapter 2. Here, we implement the Canny edge detector by actually transferring image blocks between modules, rather than using global shared data as in Chapter 2. Thus, this

[2]In Chapter 4, as the applications are communication-intensive, most threads have little computation load and thus the static thread-to-core mapping performs well.

implementation has both intensive computation and communication load. Figure 5.3 depicts the communication graph of the pipelined Canny edge detector.

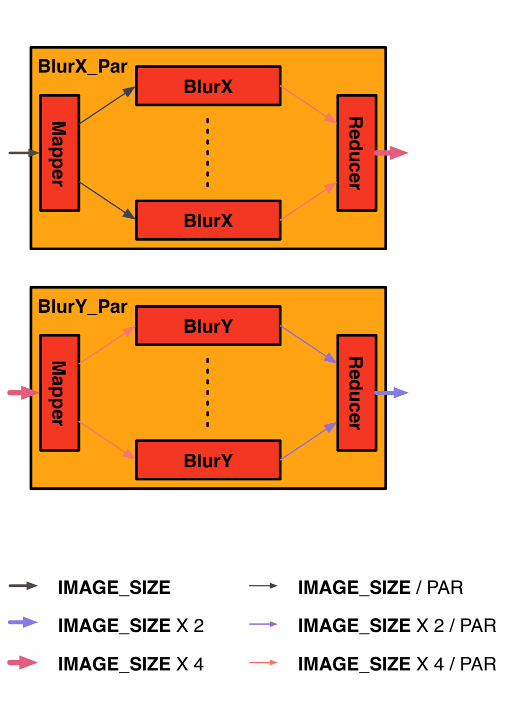


Figure 5.3: Communication graph of the pipelined Canny edge detector.

Table 5.1: Performance comparison of our proposed methods for Canny edge detector.

Host	Linux	SEG	COMM	INTG		
	Time	Speedup	Speedup	Speedup	+Speedup1	+Speedup2
12-Core Host	*50.29s*	99.16%	99.62%	**110.69%**	+11.63%	+11.11%
16-Core Host	*90.44s*	100.81%	100.32%	**102.87%**	+2.04%	+2.66%

Table 5.1 compares the performance of our proposed methods for the Canny edge detector example on the 12-core and 16-core hosts. The specification of these two workstations is presented in Table 4.3. Here, *Linux* refers to our synchronous parallel SystemC simulator using the Linux default scheduling, *SEG* refers to our segment-aware LJF optimization from Chapter 2, *COMM* refers to the parallel simulation with our communication-aware optimization from Chapter 4, and *INTG* applies our integration presented in this chapter. In the second row, *Speedup* shows the performance gain of the specific method compared to the Linux default scheduling, *+Speedup1* refers to the relative speedup of *INTG* compared

to *SEG*, and *+Speedup2* is the relative performance gain of *INTG* in comparison to *COMM*. Table 5.1 allows the two following observations:

1. **For examples with high computation and high communication load, our integration performs significantly better than the previous two methods**: In Table 5.1, while our segment-aware LJF scheduler and communication-aware optimization show similar performance to the Linux default scheduling for the Canny edge detector example, our integration of the two optimizations achieves a significant speedup of 11% on the 12-core host. Clearly, for real-world examples that have both significant computation and communication, our integration performs much better than each individual optimization from Chapter 2 and 4.

2. **Our integration shows better performance gain when most of the shared data fit into LLC**: In Table 5.1, our integration yields higher speedup on the 12-core host than on the 16-core host. Again, as discussed in Section 4.6.1, our integration reduces the costly inter-chip communication. When the parallel threads require a lot of main memory accesses on the 16-core host, the performance gain of our integration is limited[3].

[3]In the Canny edge detector example, *BlurX_Par* and *BlurY_Par* need to evenly split the processing image into slices. Thus, on the 16-core host, the image size needs to be enlarged, and part of shared images may not fit into LLC.

Chapter 6

Conclusion

In this chapter, we first summarize the contributions of this dissertation, and then discuss the future work we can explore.

6.1 Contributions

In summary, we have the following four contributions in this dissertation, namely:

1. An open-source software package for Out-of-Order PDES [56];
2. A segment-aware multi-core scheduler for SystemC PDES [57];
3. Core distance profiling on many-core and multi-core platforms [58];
4. A communication-aware thread mapping framework (Chapter 4 and 5).

6.1.1 An Open-Source Software Package for Out-of-Order PDES

In Section 1.2, we presented the Recoding Infrastructure for SystemC (RISC) [56], which implementes a dedicated SystemC compiler and an Out-of-Order PDES simulator. Other than the regular SystemC-agnostic C++ compiler, our RISC compiler is SystemC-aware, and performs segment graph construction, conflict analysis and source code instrumentation, which are necessary for the Out-of-Order PDES in the RISC simulator. OoO PDES is an advanced parallel scheduling algorithm for SystemC, which breaks the implicit temporal barriers in traditional DES and also perserves timing accuracy. The RISC simulator, which is semantics-compliant with the SystemC standard [37], allows a higher number of threads to run concurrently on parallel processing cores. Currently, the RISC package is available online at [55] as open source.

6.1.2 A Segment-Aware Multi-Core Scheduler for SystemC PDES

In Chapter 2, a *segment-aware* scheduling algorithm with optimized thread dispatching is proposed for SystemC PDES. By using the RISC compiler from Section 1.2, our scheduler [57] accurately predicts thread run times based on the static segment graph and the specific segments ahead, and then dispatches threads according to the Longest Job First (LJF) policy. The segment-aware scheduler aims at a more efficient utilization of the available processing units. Our experimental results show that our proposed segment-aware algorithm consistently speeds up SystemC PDES for both synthetic and real-world examples.

Our segment-aware multi-core scheduler covers the goal of **Reducing the Computation Time in SystemC PDES** in Section 1.5.

6.1.3 Core Distance Profiling on Multi-Core and Many-Core Platforms

In Chapter 3, we defined the *core distance* concept for multi-core and many-core platforms, and proposed a software approach [58] to optimize thread mapping for homogeneous many-core processors with distributed tag directories. For the many-core architectures using directory-based cache coherence protocols, the core-to-core communication latency depends not only on the physical placement of communicating cores on the chip, but also on the location of the distributed cache tag directory. By profiling the application and localizing the threads and the responsible tag directories, our approach can significantly reduce the on-chip communication latency on many-core platforms.

6.1.4 A Communication-Aware Thread Mapping Framework

In Chapter 4, we proposed a *communication-aware* thread mapping framework, which statically profiles communication patterns and processor architectures, and formulates the problem of thread-to-core mapping as graph partitioning. Our communication-aware framework optimizes thread mapping in order to minimize the costly inter-chip communication. This optimization meets the goal of **Mitigating the Communication Time in SystemC PDES**, and shows a consistent performance gain compared to the Linux default mapping.

Chapter 5 integrates our optimizations from Chapter 2 and 4, which covers the goal of **Decreasing the Total User Time in SystemC PDES**. The integration utilizes our communication-aware framework to minimize inter-chip communication, and dispatches ready threads on each processor according to the LJF policy.

6.2 Future Work

In addition to the research work we present in this dissertation, we would like to explore the following directions in the future.

6.2.1 Thread Mapping for Sporadic Models

As presented in Section 1.3 and 2.4.4, most of the ESL design models in SystemC are periodic. Thus, our segment-aware optimization [58] profiles the execution time of a specific segment and uses it as a prediction of its next run time. However, this is not valid for sporadic models. In future, we would like to utilize our RISC compiler to generate static load estimates, in order to improve our segment-aware scheduler for both sporadic and periodic models.

6.2.2 Extending the Thread Mapping Framework for More Platforms

In Chapter 3 and 4, we mainly study the thread mapping for homogeneous many-core processors and hierarchical multi-core processors. One direction of our future work is to extend our current thread mapping framework for more host architectures, e.g., Networks-on-Chip (NoC), Distributed Shared Memory (DSM), and Graphics Processing Units (GPU) systems. In particular, we want to add more decision criteria for these platforms to our Processor Architecture Detection Algorithm (Algorithm 7). Also, our graph partiton tool in the framework (Figure 4.5) needs to take the characteristics of these architectures into consideration, in order to automatically optimize thread-to-core mapping on these platforms.

6.2.3 Resource Contention Analysis on More Factors

In this dissertation, we focus on improving computation efficiency, on-chip and inter-chip communication overhead. However, parallel threads on the multi-core and many-core platforms contend for many other shared resources, e.g., memory access interfaces, special functional units, and on-chip cache space. In addition, the Hyper-threading technology introduced by Intel may incur extra resource contention. On multi-core and many-core platforms with hyper-threading enabled, extra logical processing cores are added to the chip. These logical processing units have their own architectural states, but share the common execution resource. This leads to additional resource contention between the hyper-threading cores. In our future work, we plan to also investigate the contention on these shared resources in our thread mapping approach.

Bibliography

[1] Tilera multi-core processors. http://www.tilera.com/products/processors.

[2] K. Andreev and H. Räcke. Balanced graph partitioning. In *SPAA*, Barcelona, Spain, June 2004.

[3] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, and A. Tal. Analyzing parallel programs with PIN. *Computer*, 43(3):34–41, March 2010.

[4] K. Bahulkar, J. Wang, N. Abu-Ghazaleh, and D. Ponomarev. Partitioning on dynamic behavior for parallel discrete event simulation. In *PADS*, Zhangjiajie, China, July 2012.

[5] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *CF*, Ischia, Italy, May 2006.

[6] L. Cai and D. Gajski. Transaction level modeling: An overview. In *CODES+ISSS*, Newport Beach, California, October 2003.

[7] Center for Embedded Computer Systems. SpecC Reference Compiler and Simulator (SCRC) 2.2. http://www.cecs.uci.edu/~specc/ftp/reference/scrc-2.2.tar.gz.

[8] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.

[9] G. Chen, F. Li, S. W. Son, and M. Kandemir. Application mapping for chip multiprocessors. In *DAC*, Anaheim, California, June 2008.

[10] W. Chen. A JPEG encoder model in SystemC. https://github.com/weiweichen/systemc-jpeg.

[11] W. Chen, X. Han, C. Chang, G. Liu, and R. Dömer. Out-of-order parallel discrete event simulation for transaction level models. *IEEE TCAD*, 33(12):1859–1872, December 2014.

[12] W. Chen, X. Han, and R. Dömer. Out-of-order parallel simulation for ESL design. In *DATE*, Dresden, Germany, March 2012.

[13] C.-L. Chou and R. Marculescu. User-aware dynamic task allocation in Networks-on-Chip. In *DATE*, Munich, Germany, March 2008.

[14] M.-K. Chung, J.-K. Kim, and S. Ryu. SimParallel: A high performance parallel SystemC simulator using hierarchical multi-threading. In *ISCAS*, Melbourne, VIC, June 2014.

[15] G. Cong and H. Wen. Mapping applications for high performance on multithreaded NUMA systems. In *CF*, Ischia, Italy, May 2013.

[16] E. H. M. da Cruz, M. A. Z. Alves, A. Carissimi, P. O. A. Navaux, C. P. Ribeiro, and J. F. Mehaut. Using memory access traces to map threads and data on hierarchical multi-core platforms. In *IPDPSW*, Shanghai, May 2011.

[17] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *HPCA*, Shenzhen, China, June 2013.

[18] T. Dey, W. Wang, J. W. Davidson, and M. L. Soffa. Characterizing multi-threaded applications based on shared-resource contention. In *ISPASS*, Austin, Texas, April 2011.

[19] T. Dey, W. Wang, J. W. Davidson, and M. L. Soffa. ReSense: Mapping dynamic workloads of colocated multithreaded applications using resource sensitivity. *TACO*, 10(4):41:1–41:25, December 2013.

[20] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: Task graphs for free. In *CODES/CASHE*, Seattle, WA, March 1998.

[21] M. Diener, E. H. M. Cruz, and P. O. A. Navaux. Communication-based mapping using shared pages. In *IPDPS*, Boston, Massachusetts, May 2013.

[22] M. Diener, E. H.M.Cruz, M. A. Z. Alves, P. O. A. Navaux, A. Busse, and H. U. Heiss. Kernel-based thread and data mapping for improved memory affinity. *IEEE Transactions on Parallel and Distributed Systems*, 27(9):2653–2666, December 2015.

[23] M. Diener, F. Madruga, E. Rodrigues, M. Alves, J. Schneider, P. Navaux, and H.-U. Heiss. Evaluating thread placement based on memory access patterns for multi-core processors. In *HPCC*, Melbourne, Australia, September 2010.

[24] R. Dömer. *System-Level Modeling and Design with the SpecC Language*. PhD thesis, University of Dortmund, Germany, April 2000.

[25] R. Dömer, W. Chen, and X. Han. Parallel discrete event simulation of transaction level models. In *ASP-DAC*, Sydney, Australia, February 2012.

[26] R. Dömer, W. Chen, X. Han, and A. Gerstlauer. Multi-core parallel simulation of system-level description languages. In *ASP-DAC*, Yokohama, Japan, January 2011.

[27] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[28] M. Fujita, I. Ghosh, and M. Prasad. *Verification Techniques for System-Level Design.* Morgan Kaufmann, San Francisco, CA, 2008.

[29] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology.* Kluwer Academic Publishers, Boston, 2000.

[30] A. Gerstlauer. Host-compiled simulation of multi-core platforms. In *RSP*, Fairfax, Virginia, June 2010.

[31] A. Gerstlauer, R. Dömer, J. Peng, and D. D. Gajski. *System Design: A Practical Guide with SpecC.* Kluwer Academic Publishers, Boston, 2001.

[32] R. Graham. Bounds on multiprocessing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[33] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC.* Kluwer Academic Publishers, 2002.

[34] B. Haetzer and M. Radetzki. A comparison of parallel SystemC simulation approaches at RTL. In *FDL*, Munich, October 2014.

[35] S. Hong, S. H. K. Narayanan, M. Kandemir, and O. Özturk. Process variation aware thread mapping for chip multiprocessors. In *DATE*, Nice, France, April 2009.

[36] Q. Hu, P. Liu, and M. C. Huang. Threads and data mapping: Affinity analysis for traffic reduction. *IEEE Computer Architecture Letters*, 15(2):133–136, June 2015.

[37] IEEE Computer Society. *IEEE Standard 1666-2011 for Standard SystemC® Language Reference Manual*, January 2012.

[38] A. Inostrosa-Psijas, V. Gil-Costa, R. Solar, and M. Marin. Load balance strategies for DEVS approximated parallel and distributed discrete-event simulations. In *PDP*, Turku, March 2015.

[39] Intel Corporation. *Intel® Xeon Phi^{TM} Coprocessor Datasheet*, June 2013.

[40] Intel Corporation. *Intel® Xeon Phi^{TM} Coprocessor System Software Developers Guide*, April 2013.

[41] International Semiconductor Industry Association. International technology roadmap for semiconductors (ITRS). `http://www.itrs.net`, 2004.

[42] J. Jeffers and J. Reinders. *Intel® Xeon Phi^{TM} Coprocssor High-Performance Programming.* Morgan Kaufmann, Waltham, MA, 2013.

[43] D. R. Jefferson. Virtual time. *TOPLAS*, 7(3):404–425, July 1985.

[44] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT*, Toronto, Canada, October 2008.

[45] L. Jin and S. Cho. SOS: A software-oriented distributed shared cache management approach for chip multiprocessors. In *PACT*, Raleigh, North Carolina, September 2009.

[46] M. Kandemir, O. Ozturk, and S. P. Muralidhara. Dynamic thread and data mapping for NoC based CMPs. In *DAC*, San Francisco, California, July 2009.

[47] G. Karypis and V. Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, August 1998.

[48] A. Kaushik and H. D. Patel. SystemC-clang: An open-source framework for analyzing mixed-abstraction SystemC models. In *FDL*, Paris, France, September 2013.

[49] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, June 2008.

[50] H.-K. Kuo, K.-T. Chen, B.-C. C. Lai, and J.-Y. Jou. Thread affinity mapping for irregular data access on shared cache GPGPU. In *ASP-DAC*, Sydney, NSW, January 2012.

[51] B.-C. C. Lai, H.-K. Kuo, and J.-Y. Jou. A cache hierarchy aware thread mapping methodology for GPGPUs. *IEEE Transactions on Computers*, 64(4):884–898, February 2014.

[52] G. Liu. Out-of-Order Parallel SystemC (OOPSC) API. `http://www.cecs.uci.edu/~doemer/risc/html_oopsc_030/index.html`.

[53] G. Liu. A hybrid thread library for efficient electronic system level simulation. Master's thesis, University of California, Irvine, Irvine, California, 2013.

[54] G. Liu, J. Park, and D. Marculescu. Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems. In *ICCD*, Asheville, NC, October 2013.

[55] G. Liu, T. Schmidt, and R. Dömer. Recoding Infrastructure for SystemC (RISC) Compiler and Simulator. `http://www.cecs.uci.edu/~doemer/risc.html`.

[56] G. Liu, T. Schmidt, and R. Dömer. RISC compiler and simulator, beta release V0.3.0: Out-of-order parallel simulatable SystemC subset. Technical Report 16-06, Center for Embedded and Cyber-Physical Systems, September 2016.

[57] G. Liu, T. Schmidt, and R. Dömer. A segment-aware multi-core scheduler for SystemC PDES. In *HLDVT*, Santa Cruz, California, October 2016.

[58] G. Liu, T. Schmidt, R. Dömer, A. Dingankar, and D. Kirkpatrick. Optimizing thread-to-core mapping on manycore platforms with distributed tag directories. In *ASPDAC*, Tokyo, Japan, January 2015.

[59] H. Liu, W.-M. Lin, and Y. Song. An efficient processor partitioning and thread mapping strategy for mesh-connected multiprocessor systems. In *SAC*, San Jose, California, April 1997.

[60] Z. Majo and T. R. Gross. A template library to integrate thread scheduling and locality management for NUMA multiprocessors. In *HotPar*, Berkeley, California, June 2012.

[61] B. B. Mandelbrot. Fractal aspects of the iteration of $z- > \lambda z(1-z)$ for complex λ and z. *Annuals of the New York Academy of Sciences*, 1980.

[62] G. Martin, B. Bailey, and A. Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology.* Morgan Kaufmann, San Francisco, CA, 2007.

[63] M. Moy. Parallel programming with SystemC for loosely timed models: A non-intrusive approach. In *DATE*, Grenoble, France, March 2013.

[64] S. Murali and G. D. Micheli. SUNMAP: A tool for automatic topology selection and generation for NoCs. In *DAC*, San Diego, California, June 2004.

[65] R. E. Nance. A history of discrete event simulation programming languages. Technical Report TR-93-21, Department of Computer Science, Virginia Polytechnic Institute and State University, 1993.

[66] M. Nanjundappa, H. D. Patel, B. A.Jose, and S. K. Shukla. SCGPSim: A fast SystemC simulator on GPUs. In *ASP-DAC*, Taipei, Taiwan, January 2010.

[67] D. Nicol and P. Heidelberger. Parallel execution for serial simulators. *TOMACS*, 6(3):210–242, July 1996.

[68] C. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *STOC*, 1988.

[69] H. D. Patel. SystemC-clang: SystemC parser using the clang front-end. `https://github.com/hdpatel/systemcclang`.

[70] D. J. Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, May 2000.

[71] G. Salvador, S. Nilakantan, B. Taskin, M. Hempstead, and A. More. Static thread mapping for NoCs via binary instrumentation traces. In *ICCD*, Seoul, Korea, October 2014.

[72] A. Sangiovanni-Vincentelli, H. Zeng, M. D. Natale, and P. Marwedel. *Embedded Systems Development: From Functional Models to Implementations.* Springer, New York, 2014.

[73] F. Sarkar and S. K. Das. Design and implementation of dynamic load balancing algorithms for rollback reduction in optimistic PDES. In *MASCOTS*, Haifa, Israel, January 1997.

[74] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge, MA, 1989.

[75] T. Schmidt. Recoding Infrastructure for SystemC (RISC) API. `http://www.cecs.uci.edu/~doemer/risc/html_risc_030/index.html`.

[76] T. Schmidt, G. Liu, and R. Dömer. Automatic generation of thread communication graphs from SystemC source code. In *SCOPES*, St. Goar, Germany, May 2016.

[77] T. Schmidt, G. Liu, and R. Dömer. Hybrid analysis of SystemC models for fast and accurate parallel simulation. In *ASP-DAC*, Tokyo, Japan, January 2017.

[78] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann. parsc: Synchronous parallel SystemC simulation on multi-core host architectures. In *CODES+ISSS*, Scottsdale, AZ, October 2010.

[79] R. Sinha, A. Prakash, and H. D. Patel. Parallel simulation of mixed-abstraction SystemC models on GPUs and multicore CPUs. In *ASP-DAC*, Sydney, NSW, January 2012.

[80] S. Sirowy, C. Huang, and F. Vahid. Online SystemC emulation acceleration. In *DAC*, Anaheim, California, June 2010.

[81] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS*, Cambridge, Massachusetts, November 2000.

[82] F. Song, S. Moore, and J. Dongarra. Feedback-directed thread scheduling with memory considerations. In *HPDC*, Monterey, California, June 2007.

[83] F. Song, S. Moore, and J. Dongarra. Analytical modeling and optimization for affinity based thread scheduling on multicore systems. In *CLUSTER*, New Orleans, LA, August 2009.

[84] S. Stattelmann, O. Bringmann, and W. Rosenstiel. Fast and accurate source-level simulation of software timing considering complex code optimizations. In *DAC*, San Diego, California, June 2011.

[85] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, 3(1):85–93, January 1977.

[86] SystemC Language Working Group (LWG). SystemC 2.3.1, core SystemC language and examples. `http://accellera.org/downloads/standards/systemc`.

[87] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystems resource sharing on datacenter applications. In *ISCA*, San Jose, California, June 2011.

[88] K. Thitikamol and P. Keleher. Thread migration and communication minimization in DSM systems. *Proceedings of the IEEE*, 87(3):487–497, March 1999.

[89] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w TeraFLOPS processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, January 2008.

[90] L. Weng and C. Liu. On better performance from scheduling threads according to resource demands in MMMP. In *ICPPW*, San Diego, California, September 2010.

[91] L. F. Wilson and W. Shen. Experiments in load migration and dynamic load balancing in SPEEDES. In *Simulation Conference Proceedings*, Washington, DC, December 1998.

[92] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*, Pittsburgh, Pennsylvania, March 2010.