AN ARCHITECTURE AND PROGRAMMING MODEL FOR EXTREMELY FINE GRAIN PARALLELIZATION

Alex Gontmakher
AN ARCHITECTURE AND PROGRAMMING MODEL
FOR EXTREMELY FINE GRAIN PARALLELIZATION

Research Thesis

Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

Alex Gontmakher

Submitted to the Senate of
THE TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY
SIVAN 5767 HAIFA MAY 2007
ACKNOWLEDGMENTS

I would like to thank Prof. Assaf Schuster for his constant encouragement and unwavering support during the course of the research. On multiple occasions, Assaf’s outstanding vision and deep understanding of the academic environment has helped me focus my efforts on things that really matter. Besides, most of what I know about writing, I have learned from Assaf during countless hours of working together on papers.

Special thanks go to Dr. Avi Mendelson, whose incredible intuition and expertise in the field have in many cases proved indispensable in my work.

Several people shared the path with me and helped make it considerably shorter. I am grateful to Sergey Polyakov, Gregory Shklover, and Vladimir Zdornov, for contributing a lot of thinking and hard work to the tools and the ideas that brought this thesis forward.

I am thankful to Dr. Evgeniy Gabrilovich, for things really too many to mention, but especially for “the gabr way” of doing and seeing things.

Many people have helped me in my work through discussions, advice and support. I am grateful to all of them: Sivan Bercovici, Alexander Berengolts, David Bernstein, Genady Beryozkin, Edward Bortnikov, Vita Bortnikov, Maxim Kovgan, Vadim Makhervaks, Bilha Mendelson, Enric Morancho, Michael Plavnik, Alex Ramirez, Mateo Valero, Vitaly Surazhsky, Tatiana Surazhsky, and Ayal Zaks.
I thank my beloved wife Julia for her love and support. She had been there when I needed it, had the patience to suffer my virtual absence when my work demanded that, and the wisdom to distinguish between the two.

Finally, I am indebted to my parents, Eva and Michael Gontmakher, for all the love, support and encouragement they have shown during these years. Words are not enough to thank them for all they have done for me.

The generous financial help of Intel Haifa Labs, the French–Israeli Fund for Cooperation in Computer Science, and of the Technion–Israel Institute of Technology is gratefully acknowledged.
# Table of Contents

## Table of Contents

vi

## List of Tables

x

## List of Figures

xi

## Abstract

1

## Abbreviations and Symbols

3

## Inthreads Instructions

7

## 1 Introduction

10

### 1.1 Motivation

10

### 1.2 Instruction Set Design

11

#### 1.2.1 Thread creation and termination

12

#### 1.2.2 Thread synchronization

13

#### 1.2.3 Thread suspension and resumption

14

### 1.3 Programming Model

16

#### 1.3.1 Consistency Model

18

#### 1.3.2 Context Switching

19

#### 1.3.3 Function Call Support

19

### 1.4 Parallelization Patterns

22

#### 1.4.1 Loop Partitioning

22

#### 1.4.2 Loop Splitting

22

#### 1.4.3 Loop Pipelining

23

#### 1.4.4 Barrier

24

#### 1.4.5 Worker Pool

24

### 1.5 Code Generation

25
1.5.1 Parallelization Approaches ..................................... 28
1.5.2 Compiler Correctness Requirements .......................... 29
1.5.3 Compiler Optimizations ........................................ 30
1.5.4 Register Allocation ................................................ 31
1.6 Microarchitecture .................................................... 32
1.6.1 Inorder Pipeline .................................................. 33
1.6.2 Out-of-Order Pipeline ........................................... 34
1.6.3 Speculative synchronization .................................... 35
1.6.4 Function Call Support ............................................ 38
1.6.5 Energy Efficient Computing .................................... 39
1.7 Conclusions .......................................................... 42

2 Intrathreads: A Technique for Parallelizing Sequential Code 43
2.1 Introduction .......................................................... 44
2.2 The Intrathreads Architecture .................................... 45
  2.2.1 Additions to the Instruction Set .............................. 47
  2.2.2 Changes in the Microarchitecture ........................... 47
  2.2.3 Inthread Instructions Overhead ............................... 49
  2.2.4 Problems and Limitations ..................................... 49
2.3 Code Transformations ............................................... 51
  2.3.1 Loop partitioning ............................................... 51
  2.3.2 Loop splitting .................................................. 54
  2.3.3 Speculative Computation ..................................... 57
  2.3.4 Worker Pool .................................................... 59
  2.3.5 Other code transformations ................................. 60
2.4 Performance Evaluation ........................................... 60
  2.4.1 164.gzip ....................................................... 61
  2.4.2 184.crafty ..................................................... 62
  2.4.3 256.bzip2 ...................................................... 62
2.5 Related Work ....................................................... 63
2.6 Conclusion .......................................................... 64

3 Speculative Synchronization and Thread Management for
Fine Granularity Threads 65
3.1 Introduction ........................................................ 66
3.2 Multithreaded Speculative Execution Model .................... 66
  3.2.1 Speculative Execution Framework .......................... 69
5.3.2 Shared Variable Identification ........................................ 123
5.3.3 Optimizations ............................................................ 124
5.4 Register Allocation .......................................................... 127
  5.4.1 Register Allocation for Concurrently-Executing Code ........ 129
  5.4.2 Spill Code Generation .................................................. 130
  5.4.3 Coalescing ............................................................... 133
  5.4.4 Register Allocation Success Guarantee .......................... 134
5.5 Microarchitecture Implementation ....................................... 135
  5.5.1 Inorder Pipeline ....................................................... 135
  5.5.2 Out-of-Order Pipeline ............................................... 137
5.6 Function Call Support ....................................................... 138
  5.6.1 Suspend/Restore Implementation .................................... 140
5.7 Evaluation ......................................................................... 144
5.8 Related Work .................................................................... 145
5.9 Conclusions ....................................................................... 146

6 Summary ................................................................. 147

Bibliography ............................................................... 152
List of Tables

3.1 Basic processor parameters .................................................. 77
3.2 Average ages and frequencies of thread-related instructions ........ 81
4.1 Parallelized region sizes ....................................................... 101
# List of Figures

1.1 Sequence of suspending and resuming multithreaded execution .......................... 15
1.2 Stages of Inthreads Code Generation .................................................. 25
1.3 Inthreads and OpenMP parallelization example ......................................... 27
1.4 Example of code that can be miscompiled by a threads-ignorant compiler ....... 30
1.5 Threading overhead comparison for Inthreads and conventional thread-based parallelization ................................................................. 34
1.6 Pipeline model for an inorder processor ............................................... 34
1.7 Pipeline model for an out-of-order processor ......................................... 35
1.8 Speculation models for single-threaded and multithreaded execution ......... 36
1.9 Inthreads microarchitecture outline ..................................................... 37
1.10 Performance of SPEC and Mediabench benchmarks with speculative execution ................................................................. 38
1.11 Processor states in the implementation of `inth.suspend` and `inth.resume` 39
1.12 Energy efficiency evaluation of the benchmarks .................................... 41

2.1 Calculating sum of lengths of strings in a linked list ................................... 44
2.2 Inthreads instruction set ........................................................................ 47
2.3 The inthreaded processor micro-architecture .......................................... 48
2.4 Partitioning of loop iterations into blocks and layers ................................ 52
2.5 Alternatives for work assignment .......................................................... 52
2.6 Results of loop partitioning ................................................................... 53
2.7 Splitting consecutive loops to execute them in a pipeline ....................... 55
2.8 Speedup achieved by loop splitting .......................................................... 57
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.14</td>
<td>Speedup of Inthreads and SMT threads-based parallelization as a function of task size</td>
<td>100</td>
</tr>
<tr>
<td>4.15</td>
<td>Energy and $ED$ results of the benchmarks</td>
<td>102</td>
</tr>
<tr>
<td>4.16</td>
<td>Processor energy consumption as a function of the IPC</td>
<td>103</td>
</tr>
<tr>
<td>4.17</td>
<td>Processor energy consumption as a function of the IPC at fetch width</td>
<td>104</td>
</tr>
<tr>
<td>4.18</td>
<td>Breakdown of energy consumption to processor components</td>
<td>104</td>
</tr>
<tr>
<td>4.19</td>
<td>Execution time as a function of the fetch policy</td>
<td>108</td>
</tr>
<tr>
<td>4.20</td>
<td>Energy and $ED$ as a function of the RS size</td>
<td>108</td>
</tr>
<tr>
<td>4.21</td>
<td>Inthreads and OpenMP parallelization example</td>
<td>109</td>
</tr>
<tr>
<td>5.1</td>
<td>Instructions defined by the Inthreads ISA</td>
<td>113</td>
</tr>
<tr>
<td>5.2</td>
<td>Inthreads Compilation Stages</td>
<td>115</td>
</tr>
<tr>
<td>5.3</td>
<td>Example of Inth-C syntax</td>
<td>116</td>
</tr>
<tr>
<td>5.4</td>
<td>Examples of differences in the definition of shared variables</td>
<td>117</td>
</tr>
<tr>
<td>5.5</td>
<td>Example of code that can be miscompiled by a threads-ignorant compiler</td>
<td>121</td>
</tr>
<tr>
<td>5.6</td>
<td>Concurrent control flow graph</td>
<td>123</td>
</tr>
<tr>
<td>5.7</td>
<td>Algorithm IDENTIFYSHAREDVARS</td>
<td>124</td>
</tr>
<tr>
<td>5.8</td>
<td>Algorithm COLLECTDEFSANDLIVE</td>
<td>125</td>
</tr>
<tr>
<td>5.9</td>
<td>Heuristic Graph Coloring Algorithm Flow</td>
<td>127</td>
</tr>
<tr>
<td>5.10</td>
<td>Register ranges for variable $a$</td>
<td>128</td>
</tr>
<tr>
<td>5.11</td>
<td>Algorithm to collect intra-thread register range conflicts</td>
<td>131</td>
</tr>
<tr>
<td>5.12</td>
<td>Data races caused by interference region spilling</td>
<td>133</td>
</tr>
<tr>
<td>5.13</td>
<td>Interference graph that causes greedy coloring to fail</td>
<td>135</td>
</tr>
<tr>
<td>5.14</td>
<td>Pipeline model for an inorder processor</td>
<td>136</td>
</tr>
<tr>
<td>5.15</td>
<td>Pipeline model for an out-of-order processor</td>
<td>137</td>
</tr>
<tr>
<td>5.16</td>
<td>Register interference resulting from a function call</td>
<td>139</td>
</tr>
<tr>
<td>5.17</td>
<td>Processor state machine for switching between multithreaded and single-threaded modes</td>
<td>141</td>
</tr>
<tr>
<td>5.18</td>
<td>Compiler actions for protecting a function call</td>
<td>143</td>
</tr>
<tr>
<td>5.19</td>
<td>Speedup of parallelization as a function of the register file size</td>
<td>144</td>
</tr>
</tbody>
</table>
5.20 Serial code slowdown as a function of the register file size . . . . . . . 145
Abstract

This work addresses one of the more important problems in today’s processor architecture: that of the amount of expressible code parallelism. It offers a new parallelism level, which takes the middle ground between the existing levels, essentially enabling multithreaded execution for code that is inherently serial with current architectures.

The proposed architecture, *Inthreads*, aims to exploit medium-to-low parallelism granularity, above that exploitable by the *instruction level parallelism* (ILP)-based mechanisms and below *thread level parallelism*-based ones. While mechanisms for utilizing the respective parallelism levels have been almost perfected, their scope is limited. On the one hand, implementation complexity and instruction dependencies prevent out-of-order processors from looking far into the instruction sequence. On the other hand, execution overheads prevent multithreading-based architectures from applying parallelization for very small tasks.

The Inthreads architecture solves the conundrum by providing an extremely lightweight threading mechanism. By sacrificing some of the flexibility, it reduces the overhead of thread management to the absolute minimum, comparable to that of instruction management under conventional architectures. To this end, Inthreads defines a programming model which requires active participation of both the hardware and software. Under the model, the threads share the processor’s resources, including the thread contexts and the architectural registers, collaboratively. The program is responsible for avoiding conflicts in concurrent accesses to the resources, while the hardware support amounts to providing special instructions for synchronization and thread management.

This work examines the implementation aspects of the architecture and studies options of its application. First, from the software point of view, the new programming model introduces a fundamental change into the compilation process. Because of the deep context sharing between the threads, the compiler processes the code of all the interacting threads together, requiring adaptation of many of the optimization algorithms. For the hardware, the fine granularity of Inthreads parallelization brings out new bottlenecks which are yet irrelevant with current architectures, re-
quiring new microarchitectural mechanisms. On a positive side, the fine granularity implies reduced complexity if the parallelized code, improving the applicability of automatic parallelization algorithms. Besides, the program-level guarantee of absence of conflicting data accesses eliminates certain kinds of interactions and significantly simplifies the processor logic.

In addition to speeding up serial code execution, the parallelism provided by the Inthreads architecture can be used to replace ILP-level architectural mechanisms. Since the programming model minimizes the hardware complexity of the implementation, the fine-grain parallelization significantly reduces the processor energy consumption, retaining the performance of single threaded execution.

In summary, the architecture presented in this work offers a new source of code parallelism, filling the granularity gap between instruction-level and thread-level parallelism. Essentially, Inthreads applies multithreading on the same level as runtime mechanisms and thus enables techniques from shared-memory programming in contexts that are considered in the domain of serial code.
## Abbreviations and Symbols

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCFG</td>
<td>Concurrent Control Flow Graph</td>
</tr>
<tr>
<td>CCR</td>
<td>Committed Conditions Register</td>
</tr>
<tr>
<td>CFG</td>
<td>Control Flow Graph</td>
</tr>
<tr>
<td>CIQ</td>
<td>Condition Instruction Queue</td>
</tr>
<tr>
<td>CMP</td>
<td>Chip Multiprocessing</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CSE</td>
<td>Common Subexpression Elimination</td>
</tr>
<tr>
<td>CST</td>
<td>Condition Speculation Table</td>
</tr>
<tr>
<td>CW</td>
<td>Condition Wait</td>
</tr>
<tr>
<td>DCache</td>
<td>Data Cache</td>
</tr>
<tr>
<td>DCE</td>
<td>Dead Code Elimination</td>
</tr>
<tr>
<td>DFA</td>
<td>Data Flow Analysis</td>
</tr>
<tr>
<td>DRF1</td>
<td>Data-Race-Free-1 Memory Consistency Model</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FPR</td>
<td>Floating Point Register</td>
</tr>
<tr>
<td>GPR</td>
<td>General Purpose Register</td>
</tr>
<tr>
<td>HB1</td>
<td>Happens Before-1</td>
</tr>
<tr>
<td>ICache</td>
<td>Instruction Cache</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction Level Parallelism</td>
</tr>
<tr>
<td>InO</td>
<td>Inorder Computing</td>
</tr>
<tr>
<td>InT</td>
<td>Inorder Computing + Inthreads</td>
</tr>
<tr>
<td>IPC</td>
<td>Instructions Per Cycle</td>
</tr>
<tr>
<td>IQ</td>
<td>Instruction Issue Queue</td>
</tr>
</tbody>
</table>
ISA — Instruction Set Architecture
OoO — Out Of Order Computing
PC — Processor Count Register
PISA — Portable Instruction Set Architecture (an extension of MIPS ISA)
RAT — Register Allocation Table
RC — Release Consistency Memory Consistency Model
ROB — Reorder Buffer
RS — Reservation Station
SC — Sequential Consistency
SMP — Symmetric Multiprocessing
SMT — Simultaneous Multithreading
SOI — Synchronization Order-1
TCU — Thread Control Unit
TID — Thread ID
TLP — Thread Level Parallelism
TMQ — Thread Management Instruction Queue
VLIW — Very Large Instruction Word
WB — Wait Buffer

Speculation Processing Symbols (Chapter 3)

\( T \) — set of threads supported by the processor
\( \text{tid}_i \) — ID of the thread that instruction \( i \) belongs to
\( i \prec j \) — instruction \( i \) precedes instruction \( j \) in the program order
\( W_t \) — sliding window of instructions of thread \( t \)
\( W_t(\tau) \) — contents of \( W_t \) at time \( \tau \)
\( S \) — set of speculative instructions
\( S(\tau) \) — set of speculative instructions unresolved at time \( \tau \)
\( C \) — set of communication events
\( c \) — \( c \) is a producer communication event
$\hat{c}$ — $c$ is a consumer communication event

$i \sim j$ — there is a chain of dependencies from $i$ to $j$

$tsv_i$ — timestamp vector of instruction $i$ (instructions that $i$ depends on)

$tsv_i(t)$ — timestamp of thread $t$ from vector $tsv_i$

$C_P(i)$ — set of instructions that affect $i$ through a chain of communication

$S_E(t)$ — the earliest currently unresolved speculation instruction of thread $t$

$C_E(t)$ — the earliest speculative communication instruction of thread $t$

$C_{SQ}(b, t)$ — the earliest instruction in thread $t$ that must be squashed because of $b$

**Compilation and Microarchitecture Correctness Symbols (Chapter 5)**

$V$ — the set of program variables

$L$ — the set of processor locations

$a^{CP}$ — $a$ is an instruction in the compiled code

$a^{CE}$ — $a$ is an instruction in the execution of the compiled code

$a^{SP}$ — $a$ is an instruction in the source code

$a^{SE}$ — $a$ is an instruction in the execution of the source code
# Inthreads Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Inth-C command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inth.start tid,addr</code></td>
<td>INTH_START</td>
<td>Starts a new thread with a given ID at a specified address.</td>
</tr>
<tr>
<td><code>inth.halt</code></td>
<td>INTH_HALT</td>
<td>Terminates the current thread. This instruction executes synchronously, guaranteeing completion of the preceding instructions.</td>
</tr>
<tr>
<td><code>inth.kill tid</code></td>
<td>INTH_KILL</td>
<td>Kills the specified thread. Unlike <code>inth.halt</code>, this instruction is asynchronous, i.e., there is no guarantee on the exact point where the thread will be terminated.</td>
</tr>
<tr>
<td><code>inth.clr cond</code></td>
<td>INTH_CLEAR</td>
<td>Clears the specified binary semaphore.</td>
</tr>
<tr>
<td><code>inth.wait cond</code></td>
<td>INTH_WAIT</td>
<td>Suspends thread execution until the semaphore is set. The value of the semaphore is automatically cleared when the thread continues.</td>
</tr>
<tr>
<td><code>inth.set cond</code></td>
<td>INTH_SET</td>
<td>Sets the value of the semaphore. If there are threads waiting on the semaphore, one of them is resumed and the value of the semaphore is cleared.</td>
</tr>
<tr>
<td><code>inth.suspend</code></td>
<td></td>
<td>Suspends the running threads and makes their context available for reading. Can be used by the operating system or by the compiler to suspend the parallel threads.</td>
</tr>
<tr>
<td><code>inth.resume tid</code></td>
<td></td>
<td>Restores the context of the running threads and continues executing in the given thread id.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Parallelism has historically been one of the most important sources of performance advancement in processor architecture. One of the fundamental tools for expressing parallelism is a thread — an independent stream of control executing a single task. Threads can be used in a wide variety of circumstances, and numerous techniques have been developed to facilitate thread-based parallelization.

An important characteristic of a threading system is the threading granularity, the minimal unit of work that can be beneficially parallelized. In general, the finer the granularity the system enables, the less constrained is the programmer in parallelizing a program. Consequently, finer grain threading systems allow parallelization in more situations than coarse-grained ones.

The choice of the target granularity of an architecture involves an inherent tradeoff. On the one hand, the finer grain systems are preferable because of the flexibility they afford to the programmer. On the other hand, the finer the parallelization granularity, the more significant part of the execution is taken by the threading system overhead.

It is possible to restrict thread semantics in order to reduce the overhead and thus make multithreading beneficial at a finer granularity. For instance, communication can be restricted to proceed in one way only [82, 74], most of the threads can be made speculative [82, 20], or just assigned to assist instruction execution rather than performing actual computation [21, 15]. These parallelization models may help alleviate some of the bottlenecks associated with sequential execution, however, speculation can be detrimental when efficiency issues, such as energy consumption, are considered. A promising direction is Transactional Memory [38, 36], which offers true parallelization and provides speculative conflict resolution in hardware, thus allowing fine grain parallelization when actual conflicts are rare.

The purpose of this work is to design a low-overhead multithreading system, Inthreads, with thread semantics approaching those of general-purpose multithreading. Such a system will allow the use of the standard shared-memory programming techniques in a fine granularity setting, essentially allowing thread-based paralleliza-
tion of code that cannot be parallelized with conventional threads.

The main principle of Inthreads design is to make the thread support an add-on functionality for the processor, requiring only minimal changes to the microarchitecture. Minimizing the changes implies that the original serialized code will run at the same speed as on the processor without Inthreads support, but speedup will be achieved when parallelization is used. The key feature of the system is a multithreading mechanism that supports a fixed number of threads and operates upon a shared register file. This carries several benefits:

- Thread setup and tear-down are extremely fast operations, incurring overhead comparable to that of a branch.

- Shared registers can be used to transfer information between threads similarly to the way shared memory locations are used in conventional multithreading, providing an extremely efficient communication medium.

- The threading mechanism, which operates within the context of an OS thread and adds little state to the OS thread context, is mostly transparent to the operating system.

In the rest of this chapter we develop the concepts behind the Inthreads architecture and programming model. Section 1.2 describes the decisions behind the Inthreads architecture and details its design. Section 1.3 describes the programming model that allows efficient implementation of all the components of an Inthreads system. In Sections 1.4 and 1.5 we describe our approach to code generation for the Inthreads architecture. In Section 1.6 we describe the hardware implementation of the architecture and its effects on performance. Finally, Section 1.7 concludes the introduction.

1.2 Instruction Set Design

The instruction set architecture of the Inthreads model consists of three mechanisms:

1. **Thread creation and termination**, consisting of instructions `inth.start`, `inth.halt`, and `inth.kill`.

2. **Thread synchronization**, consisting of instructions `inth.wait`, `inth.set` and `inth.clr`. The instructions operate on a set of single-bit *condition registers* operating as binary semaphores.

3. **Thread suspension and resumption**, consisting of instructions `inth.suspend` and `inth.resume`, and additional instructions to access the context of suspended threads.
1.2.1 Thread creation and termination

The threads operating in an Inthreads system are identified by numerical IDs, numbered from 0. Thread 0, the main thread, is always active during normal execution. Any thread can start another thread by issuing a `inth.start` instruction.

There are several reasons for supporting a fixed number of threads. First, it must be noted that since Inthread aims at a local parallelization context, the size of each parallelized region is usually relatively small both in the number of instructions and in the execution time. In such regions, the amount of parallelism expressible by threads is limited as well. In most of our benchmarks, the useful parallelization degree was 4 to 8 threads in most cases.

From the hardware point of view, the limited number of threads ensures that the size of the microarchitectural thread management mechanisms is limited by a constant. This allows the thread management logic to be implemented entirely in hardware, which is more efficient than involving software in the thread handling operations.

From the software point of view, pre-determining the number of contexts implies static assignment of computation subtasks to the threads. Given the static assignment, the compiler can be explicitly aware of the thread interactions and can potentially perform better optimization. Finally, the number of threads usable with a shared register file is limited because the number of registers available to each thread decreases with the parallelization degree, increasing the register pressure on each thread's code. At some point, the register pressure would block compiler optimization, which could harm performance and cancel the benefits of parallel execution.

The number of threads supported in a specific processor implementation is determined by a tradeoff between the desired parallelization degree, the architectural register file size and other factors such as the number of memory ports and functional units. The processor configurations evaluated in this work support up to 8 threads. Our results show that this number of threads can be supported with 64 and even as few as 32 registers. For a 128 register architecture such as IA-64 we could consider supporting 16 threads.

The `inth.start` instruction receives two parameters: the ID of the started thread and the target address of the started thread’s code. Immediately after execution of a `inth.start`, the requested thread context is activated and begins executing instructions from the specified address.

The `inth.halt` instruction deactivates the thread context of the issuing thread. The instruction has no parameters. An `inth.halt` ensures that all the previously issued instructions complete before it takes effect, and is therefore the last instruction of its thread. An `inth.halt` can be issued by any thread except for Thread 0, which must remain active at all times.

The `inth.kill` instruction deactivates a given thread context. The instruction receives a single parameter, the ID of the thread to deactivate. Since the thread that issued the `inth.kill` runs concurrently with the killed thread, the effect of the instruction is asynchronous, i.e., it is not known in advance which instructions of the killed
thread complete and which are terminated. To this end, the architecture does not specify whether the instructions that already fetched by the killed thread must be completed. The only requirement for the `inth.kill` is to be precise with respect to sequential instruction execution.

An alternative to specifying the thread ID in the `inth.start` instruction is letting the `inth.start` instruction assign thread IDs upon starting. In this case, the parameters of `inth.start` would include the register into which to write the ID instead of the thread ID itself. Using dynamically assigned IDs would relieve the compiler from the need of allocating them explicitly; moreover, it could be used to simplify the support for call frame virtualization described in Section 1.3.3.

However, the benefits of static thread ID assignment outweigh those of the dynamic one. The following reasons led us to prefer the static IDs:

- Since the code generation strategy is based on static assignment of tasks to the threads, there is little benefit in assigning thread IDs dynamically. Even if the IDs could be allocated at runtime, the compiler would have to keep track of the number of the active thread contexts, a task of roughly the same complexity as knowing which contexts are active. Besides, storing the thread IDs would increase the register pressure, which is already an issue in register-shared multithreading.

- The most important use case for thread IDs, that of the `inth.kill` instructions, is only complicated by the dynamic assignment, since with static task distribution the IDs are known to all the threads.

- The second use case for thread IDs, the suspend/resume mechanism, would become less efficient with dynamic allocation, since the thread IDs would have to be saved at the start of every parallelization sequence, wasting work in case the suspend is not done.

One complication with the static assignment of thread IDs is that the context corresponding to the required ID can be active at the time the `inth.start` is issued. This event does not imply an error in the code and occurs relatively often: it can happen if the previous parallelized region has completed but some of the `inth.halt` instructions have not executed yet and therefore some of the contexts are still active. The natural handling of such situations is to delay execution of a `inth.start` until the target context deactivates. Note that the problem would still occur, albeit less frequently, in the case of dynamic thread ID assignment: if all the available contexts are busy, the `inth.start` would have to be delayed.

1.2.2 Thread synchronization

Thread synchronization is based on a set of binary semaphores, stored in dedicated 1-bit `condition registers`. The architecture provide three instructions to manipulate the
individual condition registers: \texttt{inth.wait}, \texttt{inth.set} and \texttt{inth.clr}. All of the instructions receive a single parameter, the condition register to operate upon.

The number of condition registers is implementation-dependent. In some cases, the values of the condition registers are packed together into a single general purpose machine register. Therefore, a natural size for the condition register file is at most the number of bits in the machine word. In this work we have used 32 condition registers, although just 16 were sufficient in all the parallelization cases.

The \texttt{inth.wait} instruction checks the state of the given condition register. If it is set, the instruction immediately clears the register and proceeds. If not, the instruction is delayed until an \texttt{inth.set} is performed on the same register. The effect of executing a \texttt{inth.set} on a condition register which already has a value of 1 is no-operation, i.e., the effect of such \texttt{inth.set} is lost.

In case several threads execute \texttt{inth.wait} on the same condition, at most one \texttt{inth.wait} will proceed, and the rest will be delayed until the condition register is set again. Similarly, if several \texttt{inth.wait} by several threads are delayed, setting the condition register will wake up only one thread. The architecture does not specify the policy with which one of the threads is chosen.

In addition to accessing individual condition registers, the architecture provides instructions for bulk reading and writing of the contents of all the condition registers: \texttt{inth.cond.get} and \texttt{inth.cond.set}. These instructions are used for thread suspend/resume sequence, described below.

While we have considered the option of using counting semaphores, there are several reasons to prefer binary ones. Most importantly, as discussed above, binary semaphores allow us to pack the values of all the condition registers into a single machine register, therefore adding only a moderate overhead to thread switch sequence. In addition, binary semaphores are sufficient for most of the synchronization patterns used in our work, while hardware implementation of counting semaphores would be much more complex.

1.2.3 Thread suspension and resumption

The Inthreads architecture distinguishes two processor states: \textit{single-threaded} and \textit{multithreaded}, where in the single-threaded state, it is explicitly known that all the threads except for the main one are inactive and thus the program is allowed to manipulate the threads’ contexts. Normally, the processor switches to multithreaded state synchronously upon execution of an \texttt{inth.start} and also returns to single-threaded execution synchronously, when the threads terminate through \texttt{inth.halt} or \texttt{inth.kill} instructions. However, in some cases the program needs to switch to single-threaded mode temporarily, pausing the threads and later continuing their execution in a way transparent to the threads.

Figure 1.1 shows the sequence of events involved in suspending and resuming multithreaded execution. At any point during multithreaded execution, any of the threads can request to suspend the rest of the threads. In response, the processor pauses execution of all of the threads except the initiating one. After suspending,
Figure 1.1: Sequence of suspending and resuming multithreaded execution

The processor continues execution of the initiating thread in single-threaded mode, efficiently moving it to the context of the main thread. The first thing the thread is required to do after suspending (point 1 in the diagram), is to save the context information of all the threads, and its last action before restoring multithreaded execution (point 2) is to restore the threads’ context state. As a result of switching to the context of Thread 0 and saving all the thread contexts’ data, the execution between points 1 and 2 is indistinguishable from normal sequential execution from the program’s point of view, i.e., there are no limitations on function calls, executing parallelized code sections and so on.

The architecture provides two instructions for forceful switching between the states: `inth.suspend` freezes the threads and switches the processor to single-threaded mode and `inth.resume` switches back. The `inth.suspend` instruction receives no arguments and has the following effect. First, it stops execution of all the threads and brings all the contexts to a serializable state. After that, the processor allows access to the threads’ state, i.e., to the IPs of the threads and the values of the condition registers. As with the rest of Inthreads design, the implementation may opt to trust the software to avoid issuing the accesses improperly. Finally, it switches to single-threaded mode and continues executing instructions following the `inth.suspend`. The `inth.resume` instruction resumes execution of all of the threads from the positions stored in the IPs. The `inth.resume` receives a single parameter, the thread that the execution must resume into.

The context of multithreaded execution contains two sets of data: the IPs of the threads and the values of the condition registers. To access the IPs, the architecture provides two instructions: `inth.getip` and `inth.setip`. Both instructions receive two parameters, the thread ID and the register which is used to retrieve or store the IP, respectively. To access the condition variables, Inthreads provides two instructions: `inth.cond.get` and `inth.cond.set`. Both instructions receive a single parameter, the general purpose register that is used for storing the packed values of all the condition
The instructions above are safe for calling only during single-threaded execution mode. A suspend and resume code of thread $i$ usually contains the following instruction sequences:

<table>
<thead>
<tr>
<th>Suspend</th>
<th>Resume</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inth.suspend</code></td>
<td><code>load R10, OFF1(SP)</code></td>
</tr>
<tr>
<td><code>inth.cond.get R10</code></td>
<td><code>inth.cond.set R10</code></td>
</tr>
<tr>
<td><code>store R10, OFF1(SP)</code></td>
<td><code>load R10, OFF2(SP)</code></td>
</tr>
<tr>
<td><code>inth.getip 0, R10</code></td>
<td><code>inth.setip 0, R10</code></td>
</tr>
<tr>
<td><code>store R10, OFF2(SP)</code></td>
<td><code>inth.resume</code></td>
</tr>
<tr>
<td>· · ·</td>
<td>·</td>
</tr>
<tr>
<td><code>inth.getip 7, R10</code></td>
<td><code>inth.setip 7, R10</code></td>
</tr>
<tr>
<td><code>store R10, OFF9(SP)</code></td>
<td></td>
</tr>
</tbody>
</table>

Since the `inth.suspend` instruction switches execution context to that of Thread 0, the IP of Thread 0 would be lost if any other thread executed the `inth.suspend`. To prevent this, we hold a shadow value for the IP of Thread 0. Upon `inth.suspend`, the IP of Thread 0 is written to the shadow value, and upon `inth.resume`, the shadow value is copied back. Correspondingly, `inth.getip 0` and `inth.setip 0` always access the shadow value.

### 1.3 Programming Model

The Inthreads programming model calls for a close collaboration between the hardware and the software. The responsibility of the hardware is to provide correct implementation of the Inthreads-related instructions, and the responsibility of the software is to divide the work between the threads, assigning resources to the threads in a way that prevents access conflicts.

While Inthreads parallelization can be used in different ways, such as emulating speculative or subordinate multithreading [15], the primary application of Inthreads is true distribution of work into parallel threads. To this end, a program is divided into serial and parallelized code regions. Each of the regions corresponds to some region in the original program and performs the same computation as it. The serial regions are equivalent to the corresponding ones in the original program. The parallelized regions use threads to optimize execution of the corresponding regions in the original program.

A parallelized region is divided into code sections, where each section is associated with a specific thread ID. The code must prevent conflicts between register and memory accesses issued by different threads. To this end, private variables of the threads
must be assigned to distinct locations. Shared registers and memory locations can be explicitly used to support variable transfers, however, the program is required to avoid conflicts by protecting accesses to such variables with synchronization.

An alternative to true register sharing would be to designate special registers for communication between threads and replicate the main register file, to let each thread have a private copy of the register file. Explicit designation of communication registers would allow a distributed implementation of Inthreads, for instance, to enable the threads to execute on different cores of a multicore processor. However, there are several reasons to avoid this approach:

- The hardware implementation would not be optimal. A large part of the chip area would have to be dedicated to the parallel code execution, and would be unused during single-threaded computation or when not all of the available threads are active.

  In addition, processor hardware includes a powerful mechanism for tracking register dependencies. Dedicating special registers for communication would require the implementation to replicate this mechanism.

- The architecture would need to hardcode the numbers of private and communication registers, restricting the flexibility. As a result, it would be impossible to load balance the allocation of the registers according to the threads’ requirements. For example, one of the threads could be used for coordination of work between the other threads and would need only few registers. In our scheme of freely shared registers, the registers could be assigned to threads with higher register pressure.

  In addition, the number of required communication registers can vary widely in different programs: in some cases, multiple parameters need to be passed to threads. If the number of communication registers is hardcoded, the program would lose the ability of using as many communication registers as necessary.

  A possible benefit of explicitly designating some registers as shared is that unsynchronized concurrent accesses would be possible for them. However, the Inthreads synchronization mechanism is efficient enough so that the concurrent accesses would not provide a substantial benefit.

  Alternatively, a mechanism that designates shared registers dynamically would add even more complexity.

- The free assignment of shared registers allows the compiler to perform better optimization. First, compiler optimizations are better suited for a homogeneous register file, where every register can be used for any purpose. In addition, our scheme of register sharing allows for better register utilization. Under it, the compiler can reuse certain registers dedicated to specific tasks mandated by the calling convention, such as the stack pointer and global pointer registers.

  Besides, the values of common subexpressions could be placed in a single register and reused by all the threads. To appreciate the resulting reduction in register
Consider the case of 4 threads executing over a register file of size 32. Reusing just 4 registers, which might be insignificant for a single thread, would almost double the number of registers available to each thread in an Inthreads-parallelized region.

The Microthreads architecture [44] provides shared-register multithreading with registers designated for communication. The work does not provide a detailed implementation description, however, it is clearly designed for scalability to a large number of processors. Registers dedicated to communication aid scalability by clearly denoting which register accesses must be sent to other processor cores, however, their implementation involves considerable complexity compared to the freely shared registers of the Inthreads architecture.

1.3.1 Consistency Model

The memory consistency model of Inthreads governs both memory and register accesses. The model is designed to be simple for use and implementation. The simplicity uses the fact that Inthreads parallelization is performed in a local and tightly controlled context, with all the thread interactions explicitly visible in the code. Based on this, the consistency model allows only well-behaving programs. As shown below, this design decision eliminates many possible conflicts between instructions and thus simplifies all the components of an Inthreads implementation from the compiler to the microarchitecture.

The Inthreads consistency model is based on the data-race-free DRF1 memory consistency model [2]. DRF1 distinguishes regular data accesses from strong or synchronization instructions further identified as paired release and acquire ones, and provides ordering guarantees between the strong and the regular instructions.

The Inthreads consistency model integrates DRF1 with the instruction set architecture by explicitly designating some instructions as strong ones, stating which of them have release and acquire semantics and determining the conditions of pairing of the strong instructions. In addition, unlike DRF1, the Inthreads model manages both the memory and the register accesses.

The following instructions are designated by the Inthreads model as strong ones: `inth.start`, `inth.halt`, `inth.set` and `inth.resume` instructions have release semantics, while `inth.wait` and `inth.suspend` have acquire semantics. These instructions are inherently paired through the execution: an `inth.set` is paired with an `inth.wait` it wakes up, and an `inth.start` is paired with the first instruction of the started thread. Note that since the synchronization instructions include the condition register number, the pairing can be determined from the instruction encoding itself, i.e., can be performed immediately after instruction encoding.

We consider the integration of the consistency model with the instruction set architecture and the restriction on the set of valid programs as one of the most important features of the Inthreads architecture. As a result, Inthreads programming
model can be efficiently implemented in hardware and thus does not require a more powerful consistency framework such as [1].

DRF1 is defined on the basis of happens-before-1:

**Definition 1.1.** For each two operations $o_1$ and $o_2$ in an execution, such that $o_1$ is a release, $o_2$ is an acquire and $o_1$ is paired with $o_2$, synchronization-order-1 (SO1) relation orders $o_1$ before $o_2$. Happens-before-1 (HB1) relation is the irreflexive transitive closure of the program order and SO1.

**Definition 1.2.** Two instructions are potentially concurrent if they are not ordered by the program order, and concurrent if they are not ordered by HB1.

**Definition 1.3.** Two instruction conflict if they access the same location and at least one of them is a write. A pair of concurrent conflicting instructions forms a data race. An execution is data-race-free if it contains no data races, i.e., any two conflicting accesses in it are ordered by happens-before-1. A program is data-race-free if all its sequentially consistent executions are data-race-free.

**Definition 1.4.** Hardware obeys data-race-free-1 if the result of any execution of a data-race-free program can be obtained for the same program on sequentially consistent hardware.

### 1.3.2 Context Switching

The Inthreads architecture is designed to make the parallelized execution transparent with respect to the rest of the operating system. This is achieved since the scheduling and management of the threads is performed by the hardware within the context of a single OS thread. The only part of the operating system that is affected is the context switching logic. Naturally, the OS must suspend the threaded execution before performing the switch and resume the threads pertinent to the thread that is activated. The context information of an OS thread must be extended to include the multithreaded execution information. The amount of this information is reasonable: in our implementation it consists of 8 additional machine words, one for the condition registers and 7 for the IPs.

### 1.3.3 Function Call Support

An inherent limitation that results in from sharing of the architectural registers affects the function call mechanism. Performing function calls from within parallel code is unsafe: according to the calling convention, the called function assumes that all of the register file is in its exclusive use, and thus by saving registers to the stack it can safely access any of the architectural registers. This assumption is obviously wrong under shared-register multithreaded execution, and therefore, care must be taken with function calls in Inthreads-parallelized code.
Note that even if a virtualization mechanism is used to enable function calls from within Inthreads-parallelized code, the threads would still share the call stack. Therefore, at most one function call at a time can be executing under an active parallelized code region.

The inability of Inthreads-parallelized code to perform function calls is a significant limitation to the applicability of Inthreads parallelization. In many cases, function calls constitute an inherent component of the code and cannot be eliminated. This prevents the use of Inthreads in many circumstances, including the need for library or system calls and recursion. The problem is partly alleviated by the fact that, because of the finer parallelization granularity, the probability of encountering an unavoidable function call in an Inthreads-parallelized code region is lower than that of conventional multithreading. Still, there are cases in which the parallelized code must contain function calls, and methods of handling such cases are necessary.

In many cases, inlining can be used to free the code that needs to be parallelized from function calls. We do not discuss inlining here, as it is a standard feature in modern compilers. The only difference in the case of Inthreads might be that inlining needs to be performed more aggressively.

The problem caused by prohibition of function calls in parallelized code is less acute under Inthreads since the fine-grain parallelization can take place within a function call frame. However, some programs use such patterns of function calls as tight recursive computation; in these programs parallelization would be completely unfeasible if parallelized code cannot freely perform function calls. Nevertheless, some programs contain function calls which would need to appear in the parallelized code, but are not essential to the main, heavy computation. For such programs it is important to support function calls under limitation that only one of the threads can be performing a function call at a time. Examples of such programs are:

**Resource management** If a program keeps a buffer or a pool of some resource received from the OS, such as memory or file data, most accesses would be satisfied from the pool, performing OS calls only when the pool is exhausted. In this scheme, the majority of the pool accesses could be made from parallelized code, and switches to single-threaded mode would be rare.

**Error detection** Some programs contain error handling code, which is tightly interleaved with the computation but almost never executed. Such error handling usually involves function calls and would by itself prevent parallelization. In cases where it is the only factor that prevents the use of Inthreads, suspending would enable parallelization at very little overhead.

**Partial inlining** In some cases, a function frequently proceeds through a simple, frequently executed code path and only reaches complex code for rarely occurring inputs. Such functions would be extremely expensive or even infeasible to inline entirely, while inlining of just the frequent part would be beneficial. In such cases, we could split the function into the frequent case code which calls a separate function for the rarely executed case, using inlining for the former and
suspending for the later. This would allow us to apply Inthreads parallelization to the majority of dynamically executing code.

Examples are trigonometric functions from the *math* library.

The function call support uses the thread suspension mechanism described in 1.2.3. As with the rest of the programming model, the architecture provides the low-level operations, and the compiler must ensure execution correctness.

The Inthreads model introduces an additional function calling convention that must be employed when a function call is executed from within multithreaded code. Since the compiler knows which code belongs to single-threaded execution and which—to the multithreaded one, it is able to automatically select the appropriate calling sequence. In the extended calling sequence, the conventional register saving is preceded by thread suspension and threads’ context saving and the register restoring is followed by thread context reloading and resumption of multithreaded execution.

Certain architectural features might make function calls possible from within parallelized code. One example of such feature is the *register stack engine* in the IA-64 architecture [42]. The *register stack engine* (RSE) models a virtual stack of registers, with the top of the stack controlled by the hardware. An alloc instruction pushes a new frame on the register stack, separating the registers of the calling function from those of the called one.

Since the registers used by the called function and by the calling function are physically separated, it is possible to execute a function call from register-sharing multithreaded code. Still, several measures must be taken to support such execution:

- At most one thread must be able to execute a function call at a given time. This can be achieved by delaying alloc instructions until the active function call terminates.

- Additional registers that constitute the context of the multithreaded execution, such as the contents of the threads’ IPs and the condition registers, must be separated in the called function. A possible solution is to map these registers onto locations on the RSE. Note that this solution implies an additional constraint on the program: a logical separation of multithreaded code in the caller from that in the called function, e.g., that the values of condition registers cannot transfer information across function calls.

- The processor must deactivate threads when it evicts the registers belonging to their call frame from the hardware to the off-chip storage.

In our work, we did not use register stack virtualization since we have based the implementation on a conventional RISC architecture. However, the virtualization is a natural extension for architectures that offer register stack support.
1.4 Parallelization Patterns

Inth-C can be used for parallelization by applying work distribution transformations to certain dependency patterns in the code. Most transformations are applicable to loops or to sequences of independent statements. Section 1.5.1 discusses the use of the transformations in automatic parallelization. Below we describe the transformations, describing their applicability conditions and the organization of the parallelized code. For each transformation we show the result of its application to a code example; this demonstrates the task distribution between the threads and the use of synchronization, the details might differ for different input code.

In this work, we have applied the transformations manually, however, taking care to perform them according to the patterns described below.

1.4.1 Loop Partitioning

The loop partitioning transformation distributes the work between threads in resolution of whole iterations. For the transformation to be efficiently applicable, the iterations must be independent and the execution times of the iterations must be approximately equal (in case the iterations are not equal in length, the worker pool transformation can be used).

Depending on the number of iterations and the iteration size, the loop partitioning transformation can use different distribution of iterations to the threads: interleaving subsequent iterations or blocks of iterations between threads or assigning continuous blocks of iterations to each one of the threads. No synchronization between the threads is necessary, except for the barrier in the end to make sure all the threads have completed.

1.4.2 Loop Splitting

The loop splitting transformation distributes the computation of each iteration of a loop (or of iterations of a nested loop) between several threads. Loop splitting is applicable when the respective parts of an iteration are independent of each other but the dependent in subsequent iterations.

The transformation requires the same analysis as the standard loop splitting compiler optimization, however, instead of just executing the two parts of the split loop in sequence, it assigns them to several threads executing in parallel. No synchronization between the threads, except for the barrier in the end to make sure all the threads have completed, is necessary.

The transformation operates along the following pattern:
1.4.3 Loop Pipelining

The *loop pipelining* transformation is a more complex version of *loop splitting* for the case where each part of iteration is dependent on previous part and the same part of the previous iteration (such as dynamic programming). The transformation distributes the work similarly to the *loop splitting*, but uses synchronization to ensure dependencies in the code are met. For $n$ threads, $3n$ condition registers are necessary: 2 per thread to synchronize the data transfer and one per thread to implement the final barrier.

The transformation operates along the following pattern:

<table>
<thead>
<tr>
<th>Original</th>
<th>Parallelized</th>
<th>WORKERk:</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(i=0;i&lt;N;i++) { r=0; r=work_0(r,d0); r=work_1(r,d1); ... r=work_n(r,dn); }</td>
<td>INTH_START(1,WORKER1); ... INTH_START(n,WORKERn); #pragma inthread { for(i=0;i&lt;N;i++) { work_0; } INTH_WAIT(1); ... INTH_WAIT(n); }</td>
<td>#pragma inthread { for(i=0;i&lt;N;i++) { work_k; INTH_SET(1); INTH_HALT(); }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Original</th>
<th>Parallelized</th>
<th>WORKERk:</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(i=0;i&lt;N;i++) { r=0; r=work_0(r,d0); r=work_1(r,d1); ... r=work_n(r,dn); }</td>
<td>INTH_START(1,WORKER1); ... INTH_START(n,WORKERn); #pragma inthread { for(i=0;i&lt;N;i++) { r_0=0; r_0=work_0(r_0,d0); INTH_SET(1); INTH_WAIT(2); } INTH_WAIT(2n+1); ... INTH_WAIT(n); }</td>
<td>#pragma inthread { for(i=0;i&lt;N;i++) { r_1=r_0; INTH_SET(2); r_1=work_k(r_1,d1); INTH_SET(3); INTH_WAIT(4); ... INTH_SET(2n+1); INTH_HALT(); }</td>
</tr>
</tbody>
</table>
1.4.4 Barrier

The *barrier* transformation is applicable to code that includes several independent sequences of statements. For the transformation to be efficiently applicable, the execution times of the statements must be approximately equal and the number of the sequences must not be higher than the number of available threads (in case the number of sequences is higher, the sequences can be combined or a dynamic scheme, such as the *worker pool* described in 1.4.5 can be used).

The barrier transformation has the following effect:

<table>
<thead>
<tr>
<th>Original</th>
<th>Parallelized</th>
</tr>
</thead>
<tbody>
<tr>
<td>work0;</td>
<td>INTH_START(1,WORKER1);</td>
</tr>
<tr>
<td>work1;</td>
<td>... INTH_START(n,WORKERn);</td>
</tr>
<tr>
<td>...</td>
<td>#pragma inthread</td>
</tr>
<tr>
<td>workn;</td>
<td>{ work0;</td>
</tr>
<tr>
<td></td>
<td>INTH_WAIT(1);</td>
</tr>
<tr>
<td></td>
<td>... INTH_WAIT(n);</td>
</tr>
<tr>
<td></td>
<td>WORKER1:</td>
</tr>
<tr>
<td></td>
<td>#pragma inthread</td>
</tr>
<tr>
<td></td>
<td>{ work1;</td>
</tr>
<tr>
<td></td>
<td>INTH_SET(1);</td>
</tr>
<tr>
<td></td>
<td>INTH_HALT();</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

1.4.5 Worker Pool

The *worker pool* transformation is able to assign tasks dynamically to several threads, providing automatic load balancing. The transformation is most applicable when all the tasks perform the same computation and differ only in parameters (such as independent iterations in a loop), but can also be applied to tasks with different computations by passing the task description as one of the parameters.

Each of the worker threads operates in a loop in which it selects the next task to execute. The selection is protected by synchronization instructions. If the computation contains a serial part, that part can be factored into a coordination thread that assigns the jobs to the workers. The effect of the transformation is as follows:
1.5 Code Generation

Inthreads code generation is a two-stage process, as Figure 1.2 shows. We separate the parallelization stage, which results in code embedded with explicitly-parallelized code sections, from the machine code generation stage.

The parallelization stage can be based on different levels of abstraction, including manual parallelization, semi-automatic parallelization through a parallelization framework such as OpenMP, or automatic one.

As an interface between the parallelization and the code generation stages, we have introduced the Inth-C language, an extension of C that includes explicit paralleliza-
The syntax of Inth-C matches the definitions of the programming model. The code that belongs to each one of the threads is enclosed within a C code block and is denoted by a `#pragma inthread` command. The thread starting, termination and synchronization is performed by explicit commands in the code that correspond to the Inthreads ISA instructions: `INTH_START`, `INTH_HAL T`, `INTH_KILL`, `INTH_CLEAR`, `INTH_WAIT` and `INTH_SET`.

The compiler uses the Inthreads-related operations to deduce the information on the parallel execution. It detects code sections participating in parallelized region as those reachable by `INTH_START` commands from some point in the code. The compiler expects the parallelized region to terminate before another parallelized region is started. To this end, the control flow of the parallelized code must obey certain restrictions (most of which are checked by the compiler):

- The code blocks of all the spawned threads must be targets of `INTH_START` commands.
- Every spawned thread must either stop with an `INTH_HAL T` or contain an infinite loop (in which case, another thread must terminate it with an `INTH_KILL`). The code must ensure, using synchronization if necessary, that the parallelized region terminates within a finite time after the main thread has completed its execution of the parallel region. This requirement ensures that after a parallelized region completes, it is possible to start another parallelized region or to perform function call or return.

Note that in case of threads terminating themselves with a `INTH_HAL T`, it is possible that the threads remain active after the main thread proceeds with execution of the next region, since the final instructions of the worker threads, including the `inth.halt`, could take time to execute. It is the responsibility of the program to ensure that the execution of those instructions will cause no conflicts with the rest of the program.

- Jumps into or from the parallel code are prohibited.

An example of Inthreads parallelization can be seen in Figure 1.3. The original code, shown in Figure 1.3a, contains several independent computations. First, `compute_r` and `cond(a,r)` can be computed independently for all the iterations. Second, the accesses to `perm` are always done to different array locations and are therefore not conflicting. However, the computation of `next++` must be performed in the correct order between the iterations regardless of parallelization, and the access to `perm` must make sure to use the correct value of `next`.

It is possible to express the above dependencies through OpenMP, as Figure 1.3c shows. However, the fine granularity of the example implies that the overhead of a conventional thread-based OpenMP implementation would prevent the parallelization from achieving speedup.

The dependency pattern can be translated to the parallelized code shown in Figure 1.3b. The computation is factored into subtasks, assigning the part that needs
next = 0;
for (i=2; i<=N;i++) {
a = perm[i]->a;
r = compute_r(a);
if (cond(a,r)) {
    next++;
    perm[next]->a = a;
    perm[next]->cost = r;
}
}

#pragma omp parallel
private(a,r,next_p)
for (i=2; i<N; i++) {
a = perm[i]->a;
r = compute_r(a);
if (cond(a,r)) {
    #pragma omp ordered
    next_p=next++;
    perm[next_p]->a=a;
    perm[next_p]->cost=r;
}
}

worker1:
#pragma inthread
{
    int i;
    double a, r;
    for(i=2;i<=N;i+=2)
    {
        a = perm[i]->a;
        r = compute_r(a);
        res1 = cond(a,r);
        INTH_SET(1);
        INTH_WAIT(2);
        if (res1) {
            perm[d1]->a=a;
            perm[d1]->cost=r;
        }
    }
    INTH_SET(1);
    INTH_HALT();
}

INTH_START(1, worker1);
INTH_START(2, worker2);
#pragma inthread
{
    for (i=2, next=0; ;) {
        if (!(i <= N)) break;
        /*Interact with wkr 1*/
        INTH_WAIT(1);
        if (res1)
        d1 = ++next;
        INTH_SET(2);
        INTH_WAIT(2);
        /*Interact with wkr 2*/
        ...
    }
    /* wait for workers to complete */
    INTH_WAIT(1);
    ...
}

b) Inth-C parallelization

Figure 1.3: Inthreads and OpenMP parallelization example. The original code is part of Spec2K Mcf benchmark. The computation of comp_r and cond is independent in all the loop iterations, and the only operation that must be performed serially is the update of next.

to be serialized to a single thread, the coordinating one, and the parts that can be executed in parallel are assigned to several worker threads. Several dedicated shared variables are used for bidirectional communication between the coordinating thread and each of the workers.
1.5.1 Parallelization Approaches

Within certain limits, Inthreads can directly support some of the most important OpenMP directives, such as **parallel for** and **parallel sections**, allowing for compilation of OpenMP-based programs into Inthreads. The most important distinction is that Inthreads parallelization would create threads for each parallelized statement, and therefore would not be applicable in dynamic context.

To translate an OpenMP **parallel for** statement, the compiler would distribute the work between the threads and replicate the code of the original **for** between the threads. The visibility rules of OpenMP can be translated directly into Inth-C: in both languages, variables local to each of the threads are thread-private and variables visible to all the threads are shared. Most scheduling options can be supported: **static** scheduling can be translated directly to appropriate loop increment statements 1.4.1, **dynamic** and **guided** scheduling can be performed using the worker pool pattern 1.4.5.

The OpenMP **parallel sections** statement can be parallelized using the barrier-style Inthreads parallelization 1.4.4. In case the **parallel sections** statement includes more sections than the desired number of parallel threads, load balancing can be achieved through the worker pool pattern 1.4.5.

Note that OpenMP recognizes its granularity limit and provides a **parallel if** clause to let the programmer determine dynamically if parallelization is beneficial. Since Inthreads can perform at a finer granularity, we can use the **parallel if** clause to determine where Inthreads-based parallelization rather than conventional thread based one should be used. This distinction would fit with the respective limitations of Inthreads-based and conventional thread-based OpenMP implementation. The Inthreads-based implementation is limited in the complexity of code that can be parallelized, however, the short-running code regions for which it would be used will tend to have lower complexity. We conclude that Inthreads can complement, rather than compete with, conventional threads, widening the range of cases in which OpenMP can be applied.

In addition to OpenMP, Inthreads architecture can serve as a target for automatic parallelization. Modern compilers include advanced algorithms for dependency analysis and parallelization. The applicability of such algorithms is limited to regular code patterns with simple dependency structure. However, at the fine granularity of Inthreads application, the parallelized regions are smaller and therefore possess structure that is inherently simpler than in the general case. Therefore, the automatic compiler analysis is more likely to be applicable and thus to find parallelizable statements at the granularity that Inthreads operates at. As a result, Inthreads could extend the applicability of existing automatic parallelization algorithms to where thread-based parallelization is unprofitable.

An additional benefit of the fine granularity pertains to the data-race-free requirement of the programming model. While data-race detection is unsolvable in the general case, the automatic detection algorithms are more likely to be applicable at the fine granularity, and the resulting low complexity, of code regions that Inthreads parallelization deals with.
1.5.2 Compiler Correctness Requirements

To define the compiler correctness requirements, we use the following model of compiler operation. The compiler conceptually proceeds in two stages. During the first stage, roughly corresponding to the set of compiler optimization transformations, the compiler can introduce new temporary variables and accesses to them. It can also introduce new statements, remove or reorder existing ones. During the second stage, corresponding to register allocation, the compiler maps the statements from the set of program variables $V$ to the set of locations $L$. At each statement, variable $v_i$ can be mapped to some location $l_i$, while no two different variables can be mapped to the same location. Throughout the program, several variables can be mapped to the same location or one variable can be mapped to several locations. While practical compilers perform some transformations after the second stage, these transformations are more restricted — for instance, no new variables are introduced.

Below we use the following definitions and notations. Program statement $o$ in the original program is denoted by $o^{SP}$, and in the compiled program by $o^{CP}$. An execution point corresponding to $o^{SP}$ is denoted by $o^{SE}$, and an execution of $o^{CP}$ is denoted by $o^{CE}$. If a statement $o$ was issued by thread $T$, it is denoted by $o \in T$.

The following requirements are sufficient in order to ensure correct compilation of context-sharing multithreaded code, assuming that the source code is data-race free.

**Requirement 1.5.1.** The compiler preserves the sequential execution semantics of each thread.

**Requirement 1.5.2.** The compiler avoids introducing accesses to temporary variables in such a way as to make them shared.

**Requirement 1.5.3.** The compiler preserves the order of synchronization instructions of each thread.

**Requirement 1.5.4.** The compiler preserves the order of accesses to the shared variables with respect to synchronization operations.

**Requirement 1.5.5.** The compiler avoids introducing accesses to variables that are not temporary, except for those directly appearing in the program source.

**Requirement 1.5.6.** The compiler avoids removing accesses to shared variables that are live at some synchronization statement.

**Requirement 1.5.7.** For two concurrent statements $e_1$ and $e_2$, if variable $v_1$ is defined at $e_1$ and variable $v_2$ is live at $e_2$, the compiler does not map $v_1$ and $v_2$ to the same location at $e_1$ and $e_2$.

**Requirement 1.5.8.** For every two concurrent statements $e_1$ and $e_2$, if variable $v$ is defined at $e_1$ and live at $e_2$, $v$ must be mapped to the same location at $e_1$ and $e_2$.

To prove the correctness, we first show that the requirements ensure that the generated code is data-race-free, implying that the execution is sequentially consistent, and then claim that the execution results of the compiled code are possible under some execution of the source code.
Figure 1.4: Example of code that can be miscompiled by a threads-ignorant compiler

```
(1) INTH_START(1,THREAD1);
(2) #pragma inthread
(3) {
    (4) b = ...;
    (5) INTH_WAIT(1);
    (6) a += b;
    (7) }
(8) return a;
(9) THREAD1:
(10) #pragma inthread
(11) {
    (12) a = ...;
    (13) INTH_SET(1);
    (14) INTH_HALT();
(15) }
```

Theorem 1.1. If the compiler obeys requirements 1.5.2, 1.5.3, 1.5.4, 1.5.5 and 1.5.7, then the code resulting from compilation of a data race free program is also data race free.

The proof can be found in Section 5.2.3. Since the compiled program is data-race-free, its execution on DRF1 hardware is sequentially consistent. It remains to show that the result of any computation performed by the compiled program is feasible under execution of the source program. The proof of this claim is also found in Section 5.2.3. The proof uses the requirement 1.5.8.

1.5.3 Compiler Optimizations

As a result of execution of threads in a shared context, many implicit assumptions taken by conventional compilers do not hold for Inthreads-parallelized code. For an example, consider the code in Figure 1.4. Variable $a$ is assigned in thread $T_1$ and is later read in the main thread. If a compiler processed the code of $T_1$ separately, it would not be aware of communication through $a$, and could erroneously identify the assignment to $a$ in $T_1$ as dead code, i.e., code that produces a value never used by subsequent instructions. It would then remove the assignment to $a$, generating incorrect code.

To enable generation of correct code, the compiler must be made explicitly aware of the interactions between the threads. To this end, we represent the program structure using Concurrent Control Flow Graphs (CCFG) [33]. The CCFG used in the Inth-C compiler introduces two additional edge types. Parallel edges, which connect the outgoing edges of an INTH_START with the code of the started thread, are handled similarly to general control flow edges. Synchronization edges connect
INTH_SET instructions with corresponding INTH_WAIT ones in parallel threads. The synchronization edges are used only to analyze communication between threads.

Using the CCFG, the compiler identifies the shared variables in the program. The identification is based on the assumption that the program is free of data races: since only variables protected by synchronization can be accessed concurrently by several threads, only the variables that can be communicated along synchronization edges are considered as potentially shared. Thus, while the control flow graph analysis is conservative, the shared variable detection is fairly accurate.

The compiler optimizations are broadly divided into two classes: those dependent on the data flow information and those that are not. The data flow-insensitive optimizations are in most cases not affected by the register-shared multithreading, while the data flow-sensitive ones need adjustments in order to correctly process Inthreads-parallelized code. However, the optimizations are simplified by the fact that the compiler can assume that the program is data-race free. As a result, any two conflicting instructions must be separated by a sequence of synchronization instructions, and consequently, propagating the data flow information along the synchronization edges will capture all possible interactions between threads. Furthermore, since interactions can only occur between shared variables, only data flow items concerning these must be propagated.

Section 5.3.3 discusses adjustment of individual optimizations. In some cases, using the concurrent data flow is sufficient to ensure optimization correctness. In other cases, some of the compiler correctness requirements can be violated by the code transformations, and the data propagation must be adjusted to avoid the violation.

1.5.4 Register Allocation

The register allocation phase of the compiler is the one that is most strongly affected by Inthreads parallelization because of the register sharing. However, register allocation is also simplified due to the data-race-free assumption. Our implementation of register allocation is able to automatically handle the shared and private variables, and is based on the dominant graph coloring-based algorithm.

The coloring-based algorithm builds an interference graph that identifies which variables cannot be allocated to the same register, and analyzes the graph to assign registers to the variables. Our algorithm uses the CCFG and extends the interference graph with information on the concurrent execution. Because of the data flow propagation along synchronization edges, all the communicating uses of a shared variable in different threads are mapped to a single node in the interference graph, and therefore, the same register will be used for the variable in all the threads. In addition, conflict edges are inserted between all accesses in concurrently executing threads, and therefore, the thread-private variables of different threads use distinct register subsets.

A detailed description of the register allocation process, including additional register-related issues such as spilling, coalescing and register allocation success guarantee, are discussed in Section 5.4.1.
1.6 Microarchitecture

The hardware support for Inthreads execution must include two capabilities: concurrent execution of instructions from several threads and support for the ordering semantics of the thread-related instructions. The implementation of concurrent instruction execution is similar to that used in Simultaneous Multithreading processors. Most changes apply to the control logic of the processor, while the data path is largely unchanged.

The most important changes are made to the Fetch stage, which must be able to track IP registers for several threads and fetch their instructions in parallel. The processing of regular instructions in the pipeline is almost the same as in the processor without Inthreads support: since the instructions of different threads communicate through the same mechanisms as instructions of the same thread, instructions of different threads can be freely mixed in the processor. Exceptions are related to instructions that deal with the control flow, for instance, mispredicted branches must report the thread ID as well as the correct address.

A new pipeline stage, Wait, is dedicated for processing of some of the threads-related instructions. The implementation of the Wait stage differs between inorder and out-of-order pipelines, but the basic functionality provided by it is the same: the stage delays execution of instructions of some of the threads when they are not allowed to proceed. The primary task of the Wait stage is to take care of inth.wait instructions. To this end, the stage tracks the state of the condition registers. For every inth.wait instruction that passes through it, the stage checks the corresponding condition register, and if it is set, the stage clears the register and allows the instruction to proceed. Otherwise, the inth.wait and all the following instructions of the same thread is delayed until a inth.set instruction sets the condition.

In addition, the Wait stage is used to ensure valid starting conditions for other threading-related instructions: inth.starts, which must wait until the corresponding thread context is deactivated, and inth.suspend and inth.resume, which, as described in Section 1.6.4, can be executed only when the pipeline is emptied of in-flight instructions.

Additional details related to implementation over a specific pipeline organization are discussed in Sections 1.6.1 and 1.6.2.

Implementation of the ordering semantics is based on the assumption that the program is free of data races. To analyze implementation correctness, we determine requirements sufficient to ensure that a given implementation correctly executes Inthreads programs. The requirements are:

**Requirement 1.6.1.** The processor preserves execution semantics of each thread in isolation.

**Requirement 1.6.2.** There is a sequentially consistent execution order of strong instructions that obeys the instruction pairing order and the program orders of the threads.
**Requirement 1.6.3.** A synchronization sequence $S_{a,b}$ between two instructions $a$, $b$ is a sequence of the form

$$a = a^0, o^0_1, \ldots, o^0_k, r^0, (a^i, o^i_1, \ldots, o^i_k, r^i)_{i \in 1 \ldots n-1}, a^n = b,$$

where instruction $a^i, \ldots, r^i$ are instructions in the program order of the same thread and $r_i$ is paired with $a_{i+1}$.

For two accesses $a$ and $b$ such that $a$ writes to some location $v$, and $b$ reads from the same location, if there exists at least one synchronization sequence $S_{a,b}$ and no such synchronization sequence contains additional write accesses to $v$, then the execution of $b$ must yield the same variable written by $a$.

**Theorem 1.2.** If the processor obeys requirements 1.6.1, 1.6.2 and 1.6.3, then execution of a program with no data races will be sequentially consistent.

The proof can be found in Section 5.2.4.

Requirement 1.6.2 is satisfied by both inorder and out-of-order pipelines because the steps that instruction processing takes are not changed by the addition of Inthreads support. Requirements 1.6.2 and 1.6.3 are satisfied by inorder and out-of-order pipelines as well, but the mechanisms to achieve that differ.

As a result of the tight integration of thread management and synchronization in the processor, the Inthreads parallelization incurs a much lower overhead than that of conventional multithreading. Figure 1.5 shows an evaluation of the overhead. The benchmark compares a parallelization of a computation into two tasks into two threads using different threading mechanisms. The graph plots the parallelization speedup as a function of the task size, allowing us to gauge the overhead of the multithreading. Inthreads parallelization incurs the lowest overhead, with the overhead of regular thread-based parallelization higher by a factor from 50 (for busy waiting-based synchronization) to 1000 for thread creation for every parallelized section.

### 1.6.1 Inorder Pipeline

Figure 1.6 shows the extensions that support Inthreads execution in an inorder pipeline. All the instructions, including the thread-related ones, proceed through the same processor pipeline.

The *Issue stage* is extended to independently issue instructions of different threads, allowing instructions of one thread to proceed while instructions of another are waiting on some resource. This is necessary to allow one thread to proceed ahead of another since, unlike in an out-of-order pipeline, one instruction is not allowed to pass over another.

Requirement 1.6.2 is obeyed because the instructions of the same thread are executed in order, and all the thread-related instructions are executed at the same pipeline stage (the *inth.wait* instructions are executed on the same stage among themselves). As a result, the order in which the thread-related instructions are executed is a sequence consistent with the threads’ program orders.
Figure 1.5: Threading overhead comparison for Inthreads and conventional thread-based parallelization

Figure 1.6: Pipeline model for an inorder processor. The Inthreads-related changes are shaded.

To see that requirement 1.6.3 is obeyed, note that the \texttt{inh.wait} instructions are executed at an earlier stage than \texttt{inh.set} ones. As a result, if two instructions are ordered by a synchronization sequence, the second one will necessarily be issued after the first. Consequently, the regular processor mechanisms that track instruction dependencies will preserve the data transfer between the two instructions.

### 1.6.2 Out-of-Order Pipeline

An out-of-order (OOO) pipeline can execute instructions as soon as their resources become available, without necessarily waiting for the preceding instructions to issue. To this end, the pipeline introduces a \textit{Rename} stage, in which false dependencies are eliminated by assigning a new location for each generated value. In addition, the queue in the \textit{Issue} stage can issue ready instructions from any position.
Figure 1.7: Pipeline model for an out-of-order processor

Figure 1.7 shows the extensions that support Inthreads execution in an OOO pipeline. Since the thread-related instructions do not use regular registers, there is no need to track data dependencies between them. Therefore, they are executed on a separate sub-pipeline. To this end, after the Wait stage, all the synchronization and thread manipulation instructions are diverted to the Thread Control Unit (TCU).

Requirement 1.6.2 is a direct result of the organization of the TCU, which processes instructions in the order of their arrival. To see that requirement 1.6.3 is obeyed, note that all the release instructions are executed at the TCU, and the acquire instructions are executed at earlier stages (Wait for inth.wait, Fetch for the started threads). Thus, for any two instructions ordered by a synchronization sequence, the later instruction will be renamed at the later stage. Since an OOO pipeline establishes register dependencies at the Rename stage, the dependency between the instructions will be preserved during the execution.

1.6.3 Speculative synchronization

As a result of fine-grain parallelization, new bottlenecks appear in the microarchitecture. One of those bottlenecks involves the interaction between thread-related operations such as thread starting and synchronization and the speculative execution. In a conventional architecture that implements synchronization by means of shared memory accesses, the effect of synchronization is extremely hard to undo in the other threads. As a result, these instructions are not executed speculatively and are usually performed at the commit stage.

However, for Inthreads parallelization, the effect of not executing thread-related instructions speculatively is non-negligible. As Figure 3.2 shows, the thread-related instructions occur relatively frequently in the instruction stream of parallelized programs. Therefore, it is important to enable speculative execution of such instructions.

The primary issue that must be solved is that the effect of synchronization is
not linear in presence of multiple communicating threads. Figure 1.8 illustrates the difference. While in single-threaded execution, a misprediction event involves squashing all the following instructions in the program order, in the case of multithreaded execution a misprediction event must be propagated to additional threads that have communicated with the one containing the event. Moreover, propagation of the misprediction can be timing sensitive: for example, misprediction of event A in thread $T_0$ might propagate back to $T_0$ after the processor has finished misprediction handling of that thread and has started executing instructions on the correct path. Squashing these instructions would result in incorrect execution.

There are two solutions to the non-linearity of misprediction. The first one, involving stopping execution for as long as the event is being propagated and handled, would incur overhead comparable to that of non-speculative thread communication. The second one keeps track of all the communication between the threads and thus allows us to propagate a misprediction event to all threads in $O(1)$.

Similarly to the other components of an Inthreads system, the speculative synchronization method benefits from the Inthreads programming model. The absence of data races, which implies that any communication between regular instructions spanning different thread must be protected by synchronization instructions, allows us to consider only the thread manipulation and synchronization instructions themselves as pairs of communication instructions. This results in a considerable reduction in the number of communicating instructions that need to be considered. In addition, detecting communication between synchronization instructions is straightforward since

---

Figure 1.8: Speculation models for single-threaded and multithreaded execution
all of them are executed in the TCU.

Our implementation is based on the notion of timestamp vectors. Timestamps are attached individually to all the instructions of each thread. For each instruction, we define a timestamp vector that determines the latest instruction in the program order of every thread that can affect it. In practice, timestamp vectors need to be computed and tracked only for the synchronization and thread management instructions. The formulas for updating and performing tests on timestamp vectors can be found in Section 3.2.

Figure 1.9 outlines the microarchitecture changes necessary to support timestamp vectors and the O(1) misprediction recovery. Individual timestamps are assigned to instructions during Decode. The TCU is extended to process the timestamp vectors of all the instructions that pass through it. While the instructions arriving to the TCU apply their effect to the processor immediately, they are released from the TCU only when they become non-speculative. To this end, we introduce an additional unit, the Control Speculation Table (CST), which receives the branch instructions and receives updates on all the branch misprediction events. The CST continuously monitors the timestamp of the earliest undecided branch of each thread. The TCU compares the timestamps with the timestamp vectors of the instructions it handles to decide which instructions are speculative.

A detailed description of the implementation can be found in Section 3.4.

Our evaluation tested the effect of adding speculative execution to three different mechanisms: thread starting, thread synchronization and transfer of register values.
between threads. Implementation of the third mechanism does not need hardware support; register values can be transferred safely because of the programming model. We encode the status of the three mechanisms with three letters, using F and T to denote if speculation is enabled for the corresponding mechanism. For instance, TFF means that only thread starting can be executed speculatively.

Figure 1.10 shows the results of the evaluation. We can see that enabling speculation results in a speedup of up to 20%. Different mechanisms are important in different situations: for Mcf, the speedup is achieved by speculative synchronization, in Art — by speculative thread starting, and in Twolf the maximal speedup is achieved by combination of all the three mechanisms. A detailed discussion of the integration of speculation with Inthreads can be found in Section 3.5.

While we have discussed the integration of speculation with threads in the context of Inthreads architecture, we note that the framework is not limited to Inthreads; it may become applicable to other threading systems as finer grain parallelization is achieved.

1.6.4 Function Call Support

The implementation of `inth.suspend` and `inth.resume` includes two activities. First, execution of all the threads except the initiating one must be paused. Second, the
The out-of-order pipeline organization, used in modern high-performance microprocessors to discover and express instruction-level parallelism, are getting to the point of
diminishing returns. As a result, further increasing the sizes of instruction-tracking structures, such as the issue queue and physical register file involves a significant increase in the processor energy consumption but results in only small performance improvements.

The Inthreads architecture can provide an alternative to out-of-order pipeline organization. The Inthreads parallelization, due to its fine granularity, can operate at a level similar to that of the out-of-order pipeline. However, since the Inthreads parallelization is performed at compilation time, the control structures are much simpler and thus operate at lower energy than those of an OOO pipeline. Essentially, in this application of the architecture, Inthreads is replacing expensive runtime parallelism detection mechanisms with compile-time multithreading.

Since we intend to provide an energy-efficient alternative, the goal in this section is to provide performance comparable to that of an out-of-order processor at a lower energy consumption, rather than achieving a speedup in comparison to an OOO processor. To this end, we revert the microarchitecture to an in-order pipeline and strive for as simple implementation of microarchitectural mechanisms as possible. In Section 1.6.1 we discuss a general implementation of Inthreads over an in-order pipeline. In the rest of this section, we analyze how to minimize the complexity of specific mechanisms in order to maximize energy savings without sacrificing the performance. A detailed description of the implementation and performance analysis can be found in Section 4.3.

To simplify the Fetch stage, we note that it is not necessary to fetch instructions of all the active threads during each cycle. As results in Figure 4.7 show, there is no benefit in fetching from more than 2 threads simultaneously. Moreover, the policy of deciding from which threads to fetch is simply round-robin between all the active threads that are not stuck on an inth.wait or in Issue stage.

The organization of the Issue stage includes small inorder issue queues for each thread (in a single-threaded inorder processor, the issue queue can be implicitly contained in the latches). The simplification, compared to an OOO issue queue, follows from the fact that dependency checks need to be performed only between instructions of the same thread. Dependencies between instructions of different threads are impossible because of the programming model: since the program cannot contain data races, any dependent instructions from different threads are ordered by synchronization instructions. Therefore, a producing instruction would leave the issue stage by the time the consuming instruction arrives at issue. Figure 4.10 shows that the optimal size for the queue of each thread is 4, much smaller than that of an OOO pipeline.

The results can be seen in Figure 1.12. First note that Inthreads parallelization offers significant speedups when applied to an inorder pipeline. This can be explained by the relative inefficient instruction level parallelism exploitation of inorder organization. As a result, while the benchmarks on average execute 2x slower on an inorder processor than on out-of-order one, the addition of Inthreads to an inorder pipeline brings the performance to about the same level as that of out-of-order.

The energy consumption of an inorder processor is on average about 0.5x that
of OOO, because of the reduced complexity. The energy of an inorder processor with Inthreads is lower yet, since less static energy was wasted due to the reduced execution time. However, in the more performance-aware $ED$ and $ED^2$ metrics, we can see that while out-of-order pipeline is preferable to a plain inorder one, an inorder processor with Inthreads is better than both. A more detailed evaluation can be found in Section 4.4.4.
1.7 Conclusions

The Inthreads architecture is developed on a relatively simple motivation: utilizing an intermediate level of parallelism between that of instruction-level parallelism and conventional multithreading. We have achieved that by bringing the techniques of conventional multithreading to operate inside the processor at an extremely fine granularity.

Reducing the granularity inherently involved a tradeoff between the low overhead and the reduced functionality of the system. We chose to resolve this tradeoff by restricting the parallelized program, requiring it to avoid any conflicting accesses. This decision has benefitted all the components of Inthreads. For the compiler, it allowed automatic detection of shared variables and simple implementation of optimizations and register allocation. For the microarchitecture, it removed the need for complex resolution mechanisms of interactions between speculative instructions. Moreover, the simple implementation of Inthreads functionality has allowed the processor to save energy without sacrificing performance.

An important benefit of reducing the granularity of parallelization is that the complexity of each of the parallelized regions cannot be large. As a result, Inthreads presents an easier target for automatic tools than conventional multithreading systems: both automatic parallelization and data race detection, which are too complex to be solvable in the general case, are more likely to be applicable for Inthreads.

The rest of the paper consists of chapters that detail various aspects of Inthreads architecture and programming model. All the chapters are scientific publications, are self-contained and can be read in any order. Chapter 2 introduces the idea of the architecture, discusses software patterns that utilize the fine-grain multithreading and provides an initial performance evaluation. Chapter 3 discusses the interaction between speculative execution and thread management and synchronization, and develops efficient mechanisms to allow coexistence of both concepts in the processor. The need for these mechanisms rises from the fine granularity of Inthreads parallelization; however, our treatment has been generic and can also benefit general purpose multithreaded architectures. Chapter 4 examines the energy efficiency aspects of Inthreads. The conclusion of the chapter is that we can replace the dynamic mechanisms of parallelism discovery used by modern processors with Inthreads parallelization. As a result, we can get significant gains in energy efficiency while retaining the performance. Chapter 5 provides a comprehensive treatment on the correctness aspects of Inthreads, including both the hardware and the software components. The chapter includes a formal definition of the programming model and states correctness requirements for the compiler and the processor. It then applies the definition to prove correctness of the compiler optimizations and the microarchitecture implementation.

Finally, chapter 6 summarizes the results of this work.
Chapter 2

Intrathreads: A Technique for Parallelizing Sequential Code

Alex Gontmakher and Assaf Schuster
Techhnion – Israel Institute of Technology
\{gsasha,assaf\}@cs.technion.ac.il

Abstract

We present a technique that enables parallelization of serial code using a special kind of threads called intrathreads. Due to their granularity, which is lower than that of normal thread granularity but higher than that of ILP, intrathreads can be used to parallelize programs that were considered inherently sequential.

Intrathreads are very easy to implement on existing architectures. They involve only six additional instructions and several simple hardware changes. Intrathreads feature communication and synchronization through the register file, and therefore they involve a very small communication and synchronization overhead.

Ultimately, intrathreads can be generated by a compiler. In this work we show that straightforward code transformations lead to relatively large speedups using intrathreads. Potentially, the speedups may range from dozens to hundreds of percentage points.

\(^{1}\)Published in 6th Workshop on Multithreaded Execution, Architecture and Computation (MTEAC6), in conjunction with MICRO-35 [27]. Further results published also in [30, 28, 29].
int size = 0;
for (Node* n = first; n != 0; n = n->next)
    if (n->data != 0) size += strlen(n->data);

Figure 2.1: Calculating sum of lengths of strings in a linked list

2.1 Introduction

Modern processors perform many operations in parallel in order to achieve high performance. To realize the benefit of parallelization, high degree of resource utilization must be achieved. However, a major road block to high efficiency is posed by data and control dependencies. Long sequences of branch instructions or memory fetches are quite common in current programs. Such sequences cannot be parallelized at the instruction level, because of their dependencies. On the other hand, the computation can often be divided into small chunks that can be executed independently. Unfortunately, those chunks are usually too fine-grained to be efficiently utilized using conventional multithreaded programming techniques.

For example, consider the code in Figure 2.1, which traverses a linked list and sums up the lengths of the strings in it.

The for loop involves a lot of branching, therefore it will not be able to exhaust the processor resources. Similarly, the computation of strlen usually consists of a small loop with very little work to be done on each iteration, therefore it too will not be able to fill up the processor pipeline. Thus, by running the loop traversal concurrently with the computation of strlen will achieve higher utilization of the processor, leading to better performance.

It is very hard to write sequential code that will enable parallelization at this level, because both parts of the computation include complex branch patterns and potentially high memory latencies. On the other hand, spawning a thread for every computation of strlen, or just using a second working thread that would run these computations in parallel to the list traversal, will incur overhead of thread creation and synchronization which is by itself heavier than the code in question.

Intrathreads, or inthreads for short, provide the mechanism that can parallelize the code in granularity just suitable for the example in Figure 2.1 above. Inthreads operate entirely within the runtime context of a single thread (thus the name intrathreads).

Inthreads resemble conventional threads, but several important differences help lower their overhead to the level that makes it feasible to run very small chunks of a
program in parallel.

First, unlike the conventional multithreaded programs, instructions of all the active inthreads are issued by the processor all the time. There is no context switch that must be performed, and therefore, no overhead associated with it applies to inthreads. Second, inthreads share data in the processor’s registers, allowing for very efficient communication between them. Third, inthread support includes very efficient synchronization mechanisms.

Intrathreads can be used to implement many existing patterns of concurrent shared-memory programming. They can also be used to enhance many existing compiler optimizations by making them applicable at higher level. In this paper we show optimizations based on techniques such as thread pool, speculative execution, loop partitioning, software pipelining and others. Potentially, such transformations should be carried out by the compiler.

To evaluate the potential benefit of these optimizations, we have implemented a simulator of the architecture. The speedups achieved depend on the granularity of the parallelization and on the balance of the load across inthreads. The speedups range between dozens and hundreds of percentage points.

The rest of this paper is organized as follows. Section 2.2 describes the architectural changes necessary for current processors to support intrathreads. Section 2.3 provides examples of code transformations that can take advantage of intrathreads. Section 2.4 presents some initial experimental results with intrathreads. Section 2.5 outlines the related works. Finally, we give our conclusions in Section 2.6.

### 2.2 The Intrathreads Architecture

An *intrathread*, or *inthread*, is a context of computation in the processor that uses the bare minimum of hardware resources. Almost the only architectural state specific to a given intrathread is the Processor Count register (PC).

Intrathreads are allocated statically at compile time. When created, an intrathread receives its numeric ID, which is hardwired in the machine code. A processor has a limit on number of intrathreads that it can execute.

Intrathreads running on the same processor share the registers in the register file. In addition to minimizing the resource usage, this helps achieve several important goals.

First, similarly to the way shared memory is used for communication between regular threads, shared registers are used for communication between inthreads in a
much faster way. The forwarding mechanism of the processor can eliminate much of the propagation latency between writing and reading intrathreads. In addition, communication through registers offloads traffic from the memory subsystem, freeing it to perform the data accesses necessary for the computation task itself.

Second, having a single register file makes intrathreads relatively easy to implement, with minimal architectural changes. Furthermore, the changes are such that a program that does not use intrathreads is still able to access all of the processor’s resources, exactly as if it was executing on a non-intrathreaded processor.

Third, the intrathreads mechanism is easy to support in the OS. Due to the shared register file, the amount of additional data to store on a context switch is minimal.

Intrathreads feature a synchronization mechanism, consisting of a set of binary semaphores, which we call condition registers. Operations are defined to suspend an intrathread execution until a certain condition register is set, and to manipulate the state (set/unset) of condition registers.

The semantics of condition registers can be generalized. For instance, the scheme can be enhanced by the use of counting semaphores. In Section 2.4.3 we show that this can provide better performance by improving the load balancing. However, this generality comes with a cost in the hardware implementation. We leave this trade-off for future study.

Since the intrathreads use shared registers for communication, a memory consistency model must be defined that governs the register accesses of different intrathreads. The model we use is Release Consistency [25]. This model defines Release and Acquire semantics for special operations. Informally, a Release operation ensures that the results of all the operations preceding it in the program order will be visible to any following operation. An Acquire operation ensures that all the read operations following it in the program order will see any update which precedes it.

In our case, starting an intrathread and setting a condition register have the Release semantics, and waiting on the condition register has the Acquire semantics. Together, they ensure that if two threads synchronize through the condition registers mechanism, all the updates performed by one of them will be seen by the other one. This is similar to the strong variants of memory access operations in IA-64 [41] (see Vol.1, Ch 4.4.7), but at the register level.
2.2.1 Additions to the Instruction Set

Since the concurrently running threads in an Inthreads-parallelized section utilize different parts of the same register file, the register pressure can get quite high. Therefore, a processor supporting inthreads will usually need a high number of registers.

An instruction that starts a new inthread, `it.start`, is a special kind of branch. In addition to the branch target, it must supply the ID of the inthread to be started.

Two instructions can kill inthreads. `it.kill` terminates a given inthread in cases it is no longer needed. All the computations of the killed inthread which have not yet completed are aborted. `it.halt`, terminates the current inthread. In this case, all the instructions that have been issued prior to `it.halt` are allowed to complete.

Two instructions manage the state of condition registers: `it.cond.set` sets the condition register, and `it.cond.clr` clears it.

Finally, the `it.wait` instruction waits for a given condition register to become available.

The instructions are summarized in Figure 2.2.

Since the amount of state corresponding to each inthread is minimal, the mechanism is quite transparent to the operating system. The thread switch instruction, in addition to saving the normal processor state, must save the PCs of all the inthreads. In addition, the condition registers must be saved, which accounts to a negligible amount of data, since every condition register is one bit wide.

2.2.2 Changes in the Microarchitecture

We outline an implementation which is based on a basic 4-stage pipelined superscalar architecture. Our changes comprise a new additional stage, *Condition Wait* and some
enhancements to the existing changes. The resulting micro-architecture is depicted in Figure 2.2.2.

The fetch stage of the processor must be equipped with an array of Processor Count registers. The instructions are fetched in parallel for all the inthreads that are currently active, i.e., all inthreads whose PC values are not zero. Then, the instructions are interleaved in a round-robin fashion and are transferred to the next stage.

The task of the Condition Wait (CW) stage is to manage issuing the operations that are waiting on condition registers. To this end, the CW stage delays the \texttt{it.wait} and the following instructions until the necessary condition register is set. The functionality of this stage is very simple: it receives notification on each executed \texttt{it.cond.set} or \texttt{it.cond.clr} and enables/disables passing through the corresponding \texttt{it.wait} instructions.

Once issued, instructions can proceed almost normally through the pipeline, with the limitations described below.

Branch instructions must write the new value in the correct PC. To facilitate this, each branch instruction must keep track of the id of the inthread that issued it.

Instructions \texttt{it.start}, \texttt{it.kill}, and \texttt{it.cond.set} affect many instructions in other inthreads and are therefore very hard to undo. Therefore, they must not be executed until they are sure not to be killed. In particular, they must wait until all preceding branch instructions determine whether they will perform the jump.

Finally, to implement Release semantics, the Register Rename stage must be able to stall \texttt{it.cond.set} and \texttt{it.start} instructions until all the preceding instructions
from the same inthread have executed. To this end, the processor must keep track for each inthread of the kinds of instructions that are active.

The definition of `it.wait` requires to delay the instructions following it in the program order. This also implies the Acquire semantics with respect to register accesses.

Special care must be taken in the implementation of the Rename Buffer. In non-intrathreaded architectures, killing a writing instruction and all its following ones ensures that no dependent instructions remain. By contrast, on an intrathreaded machine, the Rename Buffer can decide to forward a value written by an instruction in one inthread to an instruction issued by another one. If the writing instruction is subsequently killed due to a branch or `it.kill`, the reading instruction will still point to a buffer entry which will never be set. To handle this situation, the Rename Buffer must re-forward the register to another, active entry.

2.2.3 Inthread Instructions Overhead

When an `it.start` instruction is executed, it writes the new PC of the corresponding inthread. The latency of the started inthread is therefore similar to the latency of a branch instruction.

The `it.start` instruction itself might sometimes be stalled to preserve Release semantics, however, the following instructions from the same inthread can continue to execute. Therefore, the `it.start` instruction involves no unnecessary delays for the inthread executing it.

Similarly, `it.kill`, `it.cond.set` and `it.cond.clr` instructions pose no extra delays for the issuing inthread except for occupying an issue slot.

To speed up the handling of condition registers, the `itCOND.set` and `it.cond.clr` instructions signal the Condition Wait stage as soon as they can be executed. Thus set and clear operations are performed already in the Register Rename stage, without waiting to be dispatched to the Execute stage.

2.2.4 Problems and Limitations

Allocating Inthread IDs. The semantics of `it.kill` require the killing thread to know the ID of the killed one. To this end, each inthread has to have an ID assigned at creation time.

In the current implementation, the IDs assigned to inthreads are fixed as a part of instruction encoding. Therefore, the compiler must keep track of the IDs of threads.
that are running at each point in the code, to know which IDs can be assigned to created inthreads.

Future extensions may consider a dynamic mechanism for thread ID assignment: \texttt{it.start} will return the ID of the created inthread into a register, and its value will be used as a parameter to the \texttt{it.kill} instruction.

**Function Calls.** Suppose that several inthreads are active and one of them calls a function. Since the other inthreads continue working and accessing registers, the called function can no longer assume that it can save any used register prior to the function call and restore it upon return. Therefore, function call cannot be performed when more than one inthread is active. To address this problem, the compiler must use inlining to eliminate function calls.

We note that in architectures with sliding register windows and dynamic inthread ID management this restriction would be relieved. Still, it is impossible to perform several function calls simultaneously by different inthreads, because this would split the stack.

**Deadlocks.** Another unpleasant side effect of using inthreads is that incorrect code might bring the processor to a state of a deadlock. Fortunately, as all the inthreads are running in the same processor core, this situation is easy to detect: all the active inthreads are waiting on some condition register.

Deadlock situations are manifestations of bugs in the generated code. Careful implementation of compiler optimizations will eliminate them.

**Speculative inthreads.** In some cases, an inthread is spawned to perform some computation speculatively. The spawning inthread later determines whether it needs the result computed by the spawnee, and kills it if necessary. If the spawnee executes an instruction that causes a fault while performing speculatively, the fault handling should be deferred until the speculative code is committed. If the speculative inthread is killed, the handling should not be performed at all. If, on the other hand, the processor runs into a deadlock, and one or more of the speculative threads are in the faulting state, then the fault(s) should be handled.

To do all the above, the compiler should distinguish between speculative and non-speculative inthreads. Thus, a speculative equivalent of \texttt{it.start} instruction must be introduced, \texttt{it.start.speculative}. The processor should keep track of which inthreads have been issued as speculative ones.
2.3 Code Transformations

In this section we show code examples and the transformations that can be applied to them in order to achieve parallelism. Due to space and time limitations, we include here only an initial, tiny subset of the potential parallelizations. A study of the full set of techniques for inthread exploitation is out of the scope of this paper, and is left for future works.

To evaluate the performance improvements offered by the parallelizations, a simulator was implemented based on the SimpleScalar software suite [11]. SimpleScalar offers the PISA architecture, which is an extension of the MIPS architecture to a 64-bit instruction word. It is able to support up to 256 registers, making it a good target architecture for inthreads (recall that due to increased register pressure inthreads cannot be implemented on machines with 32 GPRs).

SimpleScalar provides a version of gcc ported for the PISA architecture. Since the compiler algorithms for the inthreaded architecture are not yet implemented, we have manually instrumented the compiler-generated assembly code with it.* instructions to generate parallelized code. We have found writing manual inthreads-based code complex but manageable even taking into account the current lack of full-featured debugging tools.

All code transformations performed in this work do not use any information beyond what is available to the compiler.

2.3.1 Loop partitioning

In some loops, each iteration can be split into several blocks, where the computation performed on a given block depends only on the results of the corresponding block in the previous iteration. Each block may include a lot of branching and even form a variable-length loop by itself. The collection of the $i$-th blocks in their respective iterations is called the $i$-th layer. Figure 2.4 schematically illustrates this case. The arrows between blocks denote that there is a data dependency between them.

There are several ways to parallelize such code using inthreads. First, we could execute each layer in a separate inthread. Condition registers would be used to synchronize the data dependent blocks. Figure 2.5a illustrates this arrangement.

The slowest inthread will be the one performing the layer of the highest complexity. If such a layer slows down the computation considerably, we could assign several inthreads to perform its computations. An inthread running the preceding layer would then assign the blocks of work in a round-robin manner to the different inthreads.
Figure 2.4: Partitioning of loop iterations into blocks and layers

Figure 2.5: Alternatives for work assignment in parallelizing the loop in Figure 2.4. $T_0$, $T_1$, $T_2$, and $T_3$, are inthreads which take part in the resulting computation. White squares represent the blocks of code. A vertical column of blocks constitutes an iteration. All blocks performed by an inthread are contained in the gray area tagged by the inthread name.

On the other hand, if the operation of a loop is too fine-grained to be effectively split into blocks, we could combine several consequent blocks and use them as a unit of work to be distributed to inthreads. Figure 2.5c illustrates this arrangement.

To illustrate loop partitioning, consider again the code in Figure 2.1. As stated before, the traversal of the linked list can be executed in parallel with the `strlen` calls. Moreover, different `strlen` calls are independent from each other, therefore we can use several worker inthreads to execute the calls concurrently.

Figure 2.6 shows the results of optimizing of the code with inthreads. The version
with two inthreads simply uses an additional inthread for the `strlen` calls. Further speedup is obtained by using several worker threads to perform the string length calculations. With five worker inthreads, one for the linked list traversal and four additional workers, the maximal possible speedup of about 3.7 is reached at the point when the traversal becomes the computation bottleneck.

For simplicity, we give here the details of the version with two inthreads only; extending the scheme to more inthreads is straightforward.

To transfer the values between the main inthread $T_0$ and worker $T_1$, we use two condition variables $C_1$ and $C_2$ and two registers $R_{38}$, $R_{39}$. $R_{38}$ serves to receive the result from $T_1$, and $R_{39}$ serves to send a string to be processed to $T_1$.

When $T_0$ finds a non-empty string, it executes a synchronization sequence to receive exclusive access to registers $R_{38}$ and $R_{39}$. It then writes in $R_{39}$ the pointer to the string, adds the size of the previous string from $R_{38}$, signals to $T_1$, and continues searching for the next string to be processed. $T_1$ executes a similar synchronization section to return the previously computed result in $R_{38}$ and receive the next string in $R_{39}$.

Below is the pseudo-code of the loop performed by $T_0$. Initially, $C_1$ is cleared and $C_2$ is set.

```c
for (Node* n=first; n!=0; n=n->next)
    if (n->data != 0) {
        it.wait C2
        R39 = n->data;
        size += R38;
        it.cond.set C1
    }
```
Note that $T_1$ need not count the iterations. When $T_0$ receives the last result from $T_1$, it kills it by `it.kill` instruction, terminating the loop.

Below is the pseudo-code for $T_1$. Note that since inthreads do not handle subroutine calls during operation, the call of `strlen` is actually inlined.

```
for (;;) {
    it.wait C1
    str = R39;
    R38 = result
    it.cond.set C2
    result = strlen(str);
}
```

It is easy to see that the synchronization sequences are mutually exclusive and are interleaved in a lock-step fashion. Due to the Release semantics of `it.cond.set` and the Acquire semantics of `it.wait`, the new value written to $R39$ by $T_0$ will be seen in the next synchronization block of $T_1$, and the value written to $R38$ by $T_1$ will be seen in the next synchronization block of $T_0$.

### 2.3.2 Loop splitting

Consider code that contains two adjacent loops such that the $i$-th iteration of the second loop is dependent only on the iterations $0 \ldots i - 1$ of the first loop.

In this case, we can split each loop into $n$ blocks (each block containing several iterations), and spawn $n$ worker inthreads, where the $k$-th inthread performs the $k$-th block of the first loop and then the $k$-th block of the second one. When the $k$-th inthread finishes computing the $k$-th block of the first loop, it signals to the $k + 1$-th inthread to start processing the $k + 1$-th block of the first loop. Similarly, when it finishes computing the block of the second loop, it signals to the $k + 1$-th inthread to continue.

It is straightforward to extend this scheme to several adjacent loops, or to an inner loop contained in an outer loop. As the result, the consecutive loops are executed in a virtual pipeline, as shown in Figure 2.7.

For an example of such transformation, consider the following code. The function performs multiple rotations on different parts of a given array.

```
for (int i=0; i<size; i++) {
    int datum = data[i];
```
The $i$-th iteration of the while loop always writes the $i$-th entry in the buffer. Therefore, it is possible to split the inner loop to perform its iterations in a pipeline as shown in Figure 2.7.

Note that the number of iterations at which to split the while loop must be decided in advance, during compile time. This implies that profile-based optimization must be used to determine the average number of iterations that the while loop executes.

To communicate all the necessary values from $T_0$ to $T_1$, we use two condition registers $C1$ and $C2$. When $T_0$ reaches a splitting point in the loop, it signals on register $C1$. Then, $T_1$ wakes up from it.wait on $C1$ and reads all the necessary values from the registers used by $T_0$. It then signals on register $C2$ to let $T_0$ work on the next loop.

The following pseudo-code outlines the code produced for $T_0$. For simplicity, some details such as startup and termination of the worker threads, have been omitted.

```c
int j = 0;
int tmp = buf[0];
while (buf[j+1] != datum) {
    buf[j] = buf[j+1];
    j++;
}
buf[j] = tmp;
```
out: for(int i=0; i<size; i++){
    int datum = data[i];
    int tmp = buf[0];
    while(buf[j+1]!=datum){
        if(j==split0){
            it.cond.set C1
            it.wait C2
            continue out;
        }
        buf[j]=buf[j+1];
        j++;
    }
    buf[j]=tmp;
}

Below is the pseudo-code for worker $T_1$. In case that more than one worker is used, a splitting condition similar to that in the code of $T_0$ must be inserted, and synchronization with $T_2$ would be performed by condition registers $C_3$, $C_4$.

for (;;) {
    it.wait C1
    mj = j;
    mdatum = datum;
    mtmp = tmp;
    it.cond.set C2

    while (buf[mj+1] != mdatum) {
        buf[mj] = buf[mj+1];
        mj++;
    }
    buf[mj] = mtmp;
}

In some of the iterations, $T_0$ will not reach the splitting point. Therefore, the number of the outer loop iterations performed by $T_1$ will be less than the number of iterations of $T_0$. Consequently, the worker threads cannot rely on counting the number of outer loop iterations to determine when the computation has finished.

To solve the problem, we allocate an additional register for each inthread to serve as a flag denoting that $T_0$ has finished its computation. The $i$-th worker that receives
the flag, sets the flag for the $i+1$-th worker, and then performs `it.halt` to terminate itself.

A condition register is allocated to let $T_0$ wait until the worker inthreads have finished.

The total number of registers that is used in this code is 9 per worker inthread, therefore, with 5 worker inthreads, a total of 45 additional registers is used.

Figure 2.8 shows the result of parallelizing this function with varying number of inthreads, using machines with different issue width. We can see that the parallelized version with 6 inthreads is able to take advantage of issue width above 8.

2.3.3 Speculative Computation

Speculative execution of instructions is one of the more advanced techniques for improving instruction latency tolerance in modern processors. With inthreads, entire blocks of code can be executed speculatively.

Consider two consecutive blocks of code $B_1$ and $B_2$, where the decision to perform $B_2$ depends on the result of executing $B_1$. $B_2$ can be spawned in a new inthread $T_1$ which will start executing immediately. When $B_1$ is completed, then, depending on the outcome of the computation, it either waits for the result of $B_2$ or kills $B_2$ and continues.

To illustrate this technique, consider the following loop. Two strings are compared to find the first place where they do not match.

```c
while (*scan && **++scan == ***match);
```

Figure 2.8: Speedup achieved by loop splitting
Figure 2.9: Results of speculative execution. The graphs correspond to the number of iterations performed non-speculatively by $T_0$.

To parallelize the loop, we simply split it. First, several iterations are performed in $T_0$. The rest of the iterations are performed speculatively in the worker thread $T_1$.

Figure 2.9 shows the result of parallelizing the function with varying number of iterations in $T_0$. Five different input sets have been generated for the function. In the first input set, all the strings have common 4-character prefix. In the second, all strings share the same 3-character prefix, and there are 3 subsets, where all strings in each subset have a common 4-character prefix. The third input set has a common 2-character prefix with 3 subsets sharing a common 3-character prefix, etc. Every string was compared to every other string, and the average speed up was measured on each input set.

Observe that running only one iteration in $T_0$ will always degrade the performance: because of the latency of starting an intthread, $T_1$ will not be able to run in parallel with $T_0$. Running 5 and more iterations in $T_0$ will degrade the performance too, since the results of $T_1$ are almost never used. However, in cases where the execution of $T_0$ is balanced with that of $T_1$, we can see the speedup. The maximal speedup is 22%.

This example demonstrates that inthreads can be applied at extremely low granularity.

It also demonstrates that speculative execution must be used with great care. Because speculative execution involves performing extra work which may be discarded, it may sometimes lead to slowdowns.

To optimize the loop above effectively, the compiler must be able to infer the average number of the iterations that the loop will perform. This can be achieved with profile-guided optimization.
2.3.4 Worker Pool

When the iterations of a loop are completely independent of each other, we can create a worker pool of inthreads to perform the different iterations in parallel.

The main inthread $T_0$ runs the loop and produces the chunks of work to be performed. The following pseudo-code describes the loop performed by $T_0$.

```java
for(int i=0; i<size; i++) {
    it.cond.set C1
    it.wait C2
}
```

Each one of the worker inthreads runs a loop in which it takes a chunk and executes it.

```java
for (;;) {
    it.wait C1
    chunk = i
    it.cond.set C2
    // perform the work on the chunk
}
```

Note that only one set of condition variables (C1, C2) is used to coordinate the computation of $T_0$ and all the workers. In this way, $T_0$ does not have to assign chunks of work to specific inthreads, allowing for automatic load balancing between them. As a result, this technique can produce speedups linear in the number of worker inthreads.

To illustrate this technique, we use a piece of code from the initialization part in 184.crafty benchmark from the SPEC2000 suite. Since each iteration of the loop writes to a different part of array, we can compute the different iterations independently.

```java
for (int i=0; i<size; i++) {
    first_ones[i] = FirstOne(i);
    last_ones[i] = LastOne(i);
    ...
}
```

Figure 2.10 shows the speedup achieved by parallelizing this loop. Note that at extremely high processor issue width the speedup is strictly linear in the number of workers. Apparently, adding more worker inthreads could produce even greater speedups.
2.3.5 Other code transformations

We believe that the inthreads architecture is simple and general enough to implement many of the algorithms and patterns established in the field of concurrent programming. In addition, totally new techniques can be developed that utilize the low level of granularity offered by inthreads.

For example, consider a program that traverses a search tree (or performs binary search on an array).

Normally, we compare a node \( n \) with the key, and then descend to either \( n->\text{left} \) or \( n->\text{right} \), depending on the outcome. With inthreads, we can run comparisons of \( n->\text{left} \) and \( n->\text{right} \) speculatively in parallel to comparing \( n \). When the comparison of \( n \) finishes, we can decide on which of the speculative executions was indeed the necessary one and spawn two new inthreads to compute its sons speculatively. The unnecessary speculative inthread is killed.

Although one third of the work performed is wasted, three inthreads are working all the time. Thus, provided that the processor is wide enough, the total speedup of the code could reach up to 2.

2.4 Performance Evaluation

In this section we present results of inthread optimization on several benchmarks in the SPEC2000 suite. Since we have not yet implemented the compiler, we apply the inthreading transformation only to a single function in each benchmark and measure
the speedup of that function. The benchmarks selected are: 164.gzip, 184.crafty and 256.bzip.

In each benchmark, we picked the function that, according to profiling data, has the largest execution time of all the functions in the program. In 164.gzip the selected function takes about 40% of the execution time, and in 184.crafty and 256.bzip the selected function takes about 20% of the execution time.

Figure 2.11 displays the results of optimization of all the functions.

2.4.1 164.gzip

The function with the highest execution time in 164.gzip is longest_match. The function is mostly serial, however, it contains the following piece of code:

```c
for (...) {
    ...
    if (match[best_len]!=scan_end ||
        match[best_len-1]!=scan_end1||
        *match==*scan || ++match!=scan[1])
        continue;
    // computation chunk (*)
}
```

The code in the chunk (*) computes the new values of variables match and scan, and does not write any memory. Therefore, it can be executed speculatively.
To do that, a new inthread $T_1$ is spawned prior to evaluating the condition of the if statement. When $T_0$ finishes evaluating the condition, it decides whether to use the results completed by $T_1$ or to kill it.

Note that speculative execution introduces extra work. As a result, at issue width of up to 3, the processor does not have enough resources to tolerate it, resulting in worse performance than in the original version.

2.4.2 184.crafty

The function with the largest execution time in 184.crafty (a chess player program) is EvaluatePawns. The function performs position evaluation, which involves almost identical code for the black and the white sides. It is natural to run the similar evaluations in parallel by two inthreads, resulting in a maximal speedup of 2.

We note that additional optimization patterns are applicable to the code, thus further speedups can be obtained.

2.4.3 256.bzip2

The function with the largest time in 256.bzip2 is generateMTFValues. The function contains two nested loops, where the inner loop accesses the array elements in a monotonic order.

```c
for (...) {
    while (ll_i != tmp) {
        ...
        yy[j++] = tmp;
    }
    // use of j (*)
}
```

We have used the loop splitting technique from Section 2.3.2. Our optimization used 4 inthreads: $T_0$, $T_1$, $T_2$ perform different parts of the while loop, while an additional inthread $T_3$ performs the bookkeeping code (*).

Note that the maximal speedup achieved in this benchmark is below 2 when using 3 worker threads. The reason for this is the high variance in the lengths of different instances of the while loop: between 0 and 255. Consequently, the splitting of the loop does not achieve good load balancing between the workers: $T_0$ does much more computation than $T_1$ and $T_2$. 

62
To improve the balancing, shorter parts of the loop should be assigned to $T_0$. However, it is impossible for $T_0$ to make progress ahead of $T_1$ because of the boolean semantics of condition registers, thus $T_0$ would stall each time $T_1$ performs a long loop. A possible solution in such cases would be to provide a producer-consumer queue between the worker threads. However, to implement such queue efficiently, we would need stronger synchronization mechanisms, such as counting semaphores or rotating condition registers.

### 2.5 Related Work

The ideas implemented in the IA-64 architecture [41] solve the problems of control and data dependencies to some extent by using predicated execution and speculative memory access instructions. These mechanisms allow one to fold several paths of branch instructions into a single serial stream of computation that can be efficiently parallelized. However, there are many cases in which the techniques employed in the IA-64 do not provide a solution. For instance, predicated execution cannot handle jamming together two independent loops of potentially different lengths.

Simultaneous Multithreading [84] and its predecessors [46, 65] provide mechanisms for sharing processor execution resources between several threads to maximize resource utilization. Single Chip Multiprocessors [62] allow to build many processors on a single chip, sharing many of the execution resources.

Most modern processors implement these techniques to some degree: Alpha EV8, HP PA-RISC 8900, IBM Power4, Intel HyperThreading, AMD SledgeHammer.

These architectures are similar to Intrathreads in that they increase utilization of the processor’s resources by sharing them among several threads. However, the only communication and synchronization mechanism available is still the main memory. While it can be reasonably efficient for synchronizing large chunks of computation, it is prohibitively expensive for the kind of low granularity communication that Intrathreads need.

It should be noted that Intrathreads do not contradict, but rather complement all the technologies above.

Many works attempt to provide necessary communication and synchronization mechanisms to parallelize the computation.

Micro-threads [43] are similar to Intrathreads in that they share the register file between different threads and provide data synchronization. However, several differences make them unsuitable for most of the transformations presented in this paper.
Most important, micro-threads synchronize on data registers only and lack therefore the synchronization semantics necessary for complex parallelization patterns.

Multiscalar Processors [24] support grouping of instructions into coarse-grain tasks. The processor can then execute the tasks in parallel, handling dependencies between them similarly to the way it handles dependencies between individual instructions. With Intrathreads, there is no division into tasks visible to the processors. Instead, explicit synchronization instructions are used at software level, providing better flexibility and more opportunities for parallelization.

Superthreaded architecture [82] supports a special fork instruction that spawns a new thread of execution. The register file is copied on fork, allowing for efficient data transfer to the spawned thread. Each given thread can spawn an additional one only once, thus converting the execution of the program into a virtual pipeline. In contrast, Intrathreads feature free communication between the threads, allowing for applicability in larger number of cases. In addition, Intrathreads share a single register file, allowing for a much simpler implementation.

The Weld architecture [22] spawns threads during the program execution to tolerate long-latency instructions. All spawned threads are executed speculatively and merged later into the main computation. The architecture defines an additional bork instruction that is a combination of branch with a fork. Upon executing this instruction, if an execution context is available, the processor spawns a new thread at the new address. This technique improves the parallelism only to a certain degree. Indeed, the authors conclude that there is no significant benefit of using more than two execution contexts per processor.

### 2.6 Conclusion

The intrathreads architecture provides a middle layer between the instruction-level parallelism and the thread-level parallelism. They can be used to parallelize many cases where the parallelism that exists in the code cannot be exploited on instruction level, but is still too fine-grained to be exploited in the thread-level.

A lot of research is still needed in the study of intthreaded architectures. In particular, there is a great challenge in the integration of compiler transformations that will automatically produce intthreaded code.

However, our results indicate that there is great potential in using intthreads. We have shown that with common parallelization patterns performance boost can reach hundreds of percentage points.
Chapter 3

Speculative Synchronization and Thread Management for Fine Granularity Threads

Alex Gontmakher  †Avi Mendelson  Assaf Schuster  Gregory Shklover
Technion, Israel Institute of Technology  †Intel Labs, Haifa, Israel
{gsasha,assaf,gkovriga}@cs.technion.ac.il  avi.mendelson@intel.com

Abstract

Performance of multithreaded programs is heavily influenced by the latencies of the thread management and synchronization operations. Improving these latencies becomes especially important when the parallelization is performed at fine granularity.

In this work we examine the interaction of speculative execution with the thread-related operations. We develop a unified framework which allows all such operations to be executed speculatively and provides efficient recovery mechanisms to handle mis-speculation of branches which affect instructions in several threads.

The framework was evaluated in the context of Inthreads, a programming model designed for very fine grain parallelization. Our measurements show that the speedup obtained by speculative execution of the threads-related instructions can reach 25%.

†Published in 12th International Symposium on High Performance Computer Architecture (HPCA-12) [31]
3.1 Introduction

Modern processors use a complex set of ILP-enhancing mechanisms, such as speculation and multithreaded execution. When combined carefully, these mechanisms complement and amplify each other [81]. However, the mechanisms may interfere and hurt each other’s performance. Moreover, the lower the locking granularity, the more significant the performance impact of synchronization [86].

In this work, we investigate the interaction of control speculation with thread management and synchronization instructions. We develop a general framework that enables speculative execution of such instructions and provides an efficient mechanism for misspeculation recovery. The framework, based on identifying and keeping track of the interactions between instructions of different threads, provides a common mechanism that handles speculative execution of synchronization, thread starting and even thread termination. The case of thread termination involves peculiar side effects due to its reverse effect on speculation, as described in Section 3.3.1.

We apply the framework to Inthreads [30], a lightweight threading model that allows parallelization at a resolution comparable to that of speculative execution. Due to the low parallelization granularity, Inthreads programs are highly sensitive to synchronization latency and benefit from the speculative execution of thread-related operations.

The rest of this paper is organized as follows. In Section 3.2 we develop the framework of speculative execution of thread management and synchronization instructions. In Section 3.3 we apply the framework to Inthreads. Section 3.4 discusses the implementation and Section 3.5 presents the results of the experimental evaluation. Finally, Section 3.6 describes the related work and Section 3.7 concludes the paper.

3.2 Multithreaded Speculative Execution Model

Modern processors use a variety of mechanisms for improving the computation parallelism. Two such mechanisms are speculative execution, intended to improve instruction-level parallelism, and multithreading, aimed at thread-level parallelism. Usually, these mechanisms operate at different levels and are orthogonal. However, in case of low granularity parallelization, the synchronization may stand in the way of efficient speculative execution.

For an example, consider the program in Figure 3.1. When the program is executed serially, branch prediction may allow the processor to execute as many iterations
in parallel as the hardware can accommodate. However, in the case of parallelized code, the presence of synchronization limits the number of iterations that can be issued speculatively: since a mutex affects execution of other threads’ instructions, it is dangerous to enter the mutex before the if has been resolved. As a result, in each thread, the if must be resolved before the next if can be issued. Therefore, the number of iterations that can proceed in parallel is at most one per active thread, potentially leading to lower ILP than that of the serial, but speculative, code.

To realize the potential of both speculation and multithreading, we must enable speculative execution of instructions that involve communication between threads, such as interactions between instructions involved in a synchronization or between instructions accessing a shared variable. In order to recover from misspeculation, we must keep track of all the communication events, and take care of all the instructions, from all the threads, that have been affected by the misspeculated instruction.

Examples of communication events are transfer of a value through a shared variable or an interaction between synchronization instructions. For the purposes of this section, it suffices to note that a communication event consists of a pair of interacting instructions, a producer one providing some information to a consumer one.

The speculation model of sequential computation is linear, as shown in Figure 3.2a: if an instruction is misspeculated, all the following instructions are squashed. In contrast, the speculation model of multithreaded computation is non-linear. Consider the execution in Figure 3.2b with threads $T_0$, $T_1$ and $T_2$, three speculation points $A$, $B$ and $C$, two communication events, 1 and 2, between $T_0$ and $T_1$, two events, 3 and 4, between $T_1$ and $T_2$, and a communication 5 from $T_2$ to $T_0$. It is impossible
to arrange the instructions so that all the instructions following a speculation point are those affected by it. For example, A and B are independent, and neither of them should precede the other.

Another observation is that misspeculation recovery is timing sensitive. Consider the misspeculation recovery of B. The misspeculation propagates to T2 along event 3, squashing the instructions following the consuming instruction of event 3. As a result, the producing instruction of event 4 is squashed, and therefore, the consuming instruction of 4 must be squashed as well. However, that instruction may have been already squashed since it follows B in the program order of T1, and the pipeline may already contain the instructions for the correct execution path after B. In that case, the processor would squash instructions on the correct execution path. The problem becomes more acute when misspeculation is propagated through longer sequences of events, such as the sequence of 2, 3 and 5 that would result from misspeculation of A.

A straightforward, albeit expensive, approach to this problem would require stopping the front-end during the propagation of branch misprediction. Our approach, described below, performs the recovery in one step without the need for propagation.
3.2.1 Speculative Execution Framework

The definitions in this section are conceptually similar to the happens-before relation defined by Lamport [50] and the related notion of vector clocks by Mattern [60]. These works are developed for distributed processors over a not necessarily in-order medium. Furthermore, the feasibility of bounded timestamps is shown in [19, 34], although system is not truly distributed and thus is simpler.

Our framework is defined over a set of threads rather than processors, and directly models the control speculation and the interactions between instructions. The framework can be applied to an architecture by identifying the possible interactions, and handling them as described in Section 3.2.2. In Section 3.3.1 we apply the framework to Inthreads.

Let T be the set of threads supported by the processor. We model the program as a set of instructions organized in program orders for each thread \( t \in T \). We denote that instruction \( i \) belongs to the program order of thread \( t \) by \( tid_i = t \). The program order defines a full order for instructions of each thread. We denote that instruction
i precedes instruction j in the program order by i ≺ j.

Execution of the program proceeds through sliding windows of instructions. There is one sliding window \( W_t \) for each thread \( t \). \( W_t \) operates as a FIFO: instructions are introduced at one end and retired at another. \( W_t(\tau) \) denotes the contents of \( W_t \) at time \( \tau \) and \( W(\tau) = \bigcup_{t \in T} W_t(\tau) \).

Instructions are classified into plain and speculation point ones. The speculation point instructions are introduced as unresolved. When resolved, a speculation point instruction can be found to be misspeculated, in which case all the following instructions must be squashed. We denote the set of speculation point instructions by \( S \) and the set of currently unresolved ones by \( S_?(\tau) \). Consequently, the set of speculative instructions at time \( \tau \) is \( \{i \in W(\tau) : \exists b \in S_?(\tau) ts v_i(tid_b) > ts_b\} \).

Some instructions participate in communication events. A communication event \( c \) consists of two instructions, the producing one \( \dot{c} \) and the consuming one \( \ddot{c} \) \( (c = (\dot{c}, \ddot{c})) \). \( c \) introduces a dependency: if \( \dot{c} \) is squashed, then \( \ddot{c} \) must be squashed as well. \( i \rightarrow j \) denotes that \( j \) depends on \( i \) through some chain of dependencies. We denote the set of communication events by \( C \).

In the example in Figure 3.3, \( S = \{i_2, i_5, i_7, j_3\} \), \( S_?(\tau) = \{i_5, i_7, j_3\} \), and \( C = \{c_1 = (i_3, j_4), c_2 = (k_1, j_2), c_3 = (j_5, k_6), c_4 = (j_7, k_8)\} \).

Each instruction \( i \) has a timestamp, \( ts_i \). The timestamps are assigned independently for each thread, in such a way that \( ts_i < ts_j \iff i < j \). A timestamp vector, \( ts v_i \), defines the latest instruction in each thread \( t \) that \( i \) depends on:

\[
    ts v_i(t) = \begin{cases}
        \max_{c \in C_P(i)} ts_c, & t \in T \setminus \{tid_i\} \\
        ts_i, & t = tid_i
    \end{cases}
\]

where \( C_P(i) \) is the set of instructions that affect \( i \) through a chain of communication events \( c_1, \ldots, c_n \):

\[
    C_P(i) = \left\{ \begin{array}{l}
    c : \exists c_1, \ldots, c_n \in C, c = c_1 \land \ddot{c}_n \prec i \land \\
    \forall j \in \{1, \ldots, n-1\} (\dot{c}_j \prec \dot{c}_{j+1})
    \end{array} \right\}.
\]

The value of \( ts v_j \) can be used to determine if \( j \) depends on a given instruction \( i \):

\[
    i \rightarrow j \iff ts_i \leq ts v_j(tid_i)
\]

When a speculation point instruction \( b \) is found to be mispredicted, we need to squash all the instructions \( \{i \in W(\tau) : ts v_i(tid_b) > ts_b\} \). Consequently, the set of speculative instructions at time \( \tau \) is \( \{i \in W(\tau) : \exists b \in S_?(\tau) ts v_i(tid_b) > ts_b\} \).
In the example in Figure 3.3, \( tsv_{j_4} = [3, 4, 1] \), \( tsv_{j_2} = [\emptyset, 2, 1] \), \( tsv_{k_6} = [3, 5, 6] \) and \( tsv_{k_8} = [8, 7, 1] \). If \( j_3 \) is mispredicted, checking the \( tsv \) reveals that both \( i_8 \) and \( k_6 \) must be squashed. The speculative instructions (i.e., those that might be squashed) are \( i_8 \) in \( T_0 \), \( j_4 \ldots j_7 \) in \( T_1 \) and, because of communication event \( c_3 \), \( k_6 \ldots k_7 \) in \( T_2 \). Note that \( j_2 \) is not speculative, since no unresolved speculation point instructions precede \( k_1 \).

In the remainder of this section, we will develop efficient ways for computing the set of speculative instructions and for performing misspeculation recovery.

First, if an instruction \( i \) preceded by \( i' \) does not take part in a communication event, then \( tsv_i(t) = tsv_{i'}(t) \) for any thread \( t \neq tid_i \). Therefore, in recovering from a misspeculation from a speculation point \( b \), the earliest squashed instruction in every thread except \( tid_b \) must be an instruction that takes a consumer part in a communication. As a result, it is sufficient to just scan the instructions in \( C \cap W(\tau) \) in order to determine which instructions must be killed as a result of misspeculation. In addition, it is sufficient to compute \( tsv \) only for the communicating instructions.

Second, we note that since the timestamps are monotonically increasing in each thread, we need to scan only the earliest unresolved speculation point in each thread in order to determine which instructions are speculative.

To summarize, the determination of speculative instructions and the misspeculation recovery are performed in the following way. First, we compute the set of earliest speculation points in each thread by scanning all the currently unresolved speculation point instructions:

\[
S_E(t) = t_b : b \in S_\tau(\tau, t) \land \nexists b' \in S_\tau(\tau, t) b' < b
\]

Using \( S_E \), we can determine whether a given communication instruction \( \hat{c} \) is speculative:

\[
\text{spec}_{\hat{c}} = \exists t S_E(t) < tsv_{\hat{c}}(t)
\]

Now we can compute the timestamps of the earliest speculative communication instructions in each thread:

\[
C_E(t) = t_{\hat{c}} : \hat{c} \in C \cap W_\tau(\tau) \land \text{spec}_{\hat{c}}
\]

With \( S_E \) and \( C_E \), we can determine whether instruction \( i \) is speculative by checking that \( ts_i \geq S_E(tid_i) \lor ts_i \geq C_E(tid_i) \). The speculative instructions may not be retired even if their execution is completed.

71
When a speculation instruction $b$ is found to be misspeculated, we determine the set of instructions to be squashed in the following way. For each thread, we scan the communication instructions to determine the earliest squashed one:

$$c \in C \cap W_t(\tau) \land$$

$$C_{SQ}^b(b, t) = tsv_1^c : tsv_1^c(tid_b) > ts_b \land$$

$$\nabla_1 \land tsv_1^c(tid_b) > ts_b$$

Instruction $i$ must be squashed during the recovery of $b$ if $ts_i \geq C_{SQ}^b(b, tid_i)$.

Finally, the computation of $tsv$ is performed according to its definition by scanning the instructions in $C \cap W(\tau)$.

Returning to the execution in Figure 3.3, the computations would proceed in the following way: $S_2 = [5, 3, \emptyset], C_2 = [8, 4, 6]$, exactly as in the original definitions. If $j_3$ is misspeculated, then $C_{SQ}^b(j_3) = [8, 4, 6]$, implying that the instructions squashed in addition to the instructions following $j_3$ in $T_1$ are $i_8$, $k_6$ and $k_7$.

### 3.2.2 Handling Communication Instructions

For each instruction $i$ that can potentially take part in a communication event, we must choose one of the two alternative implementations:

- **Issue $i$ speculatively.** In this case, we need to dynamically determine the instructions that participated in a communication with $i$. The $tsv$ for these instructions must be updated accordingly to enable proper recovery.

- **Delay $i$ until it becomes non-speculative.** Naturally, the instruction that participates in communication with $i$ will not be able to issue before $i$. On the positive side, squashing of $i$ cannot affect instructions from other threads, and thus the $tsv$ of the instruction receiving the communication need not be updated.

### 3.3 Speculation in the Inthreads Model

The Inthreads model is based on a fixed number of threads running over shared registers in the context of a single SMT thread. The model provides an extremely lightweight threading mechanism: the fixed number of threads allows for thread management to be performed entirely in hardware. In addition, the shared registers provide a straightforward and efficient communication mechanism: a value can be
transferred by writing it into a register in one thread and reading it from the same
register in another.

Each thread has a thread ID (tid) which is determined at thread creation. The
main thread, identified by tid = 0, is always active, while other threads can be started
and terminated on demand. Three instructions control the starting and stopping of
threads: inth.start, inth.halt and inth.kill. inth.start creates a new thread with a given
tid at a given address. To terminate itself, a thread issues an inth.halt instruction.
inhalt is executed synchronously, guaranteeing that all the instructions preceding
it will complete. One thread can kill another by issuing an inth.kill instruction.

The synchronization mechanism consists of a set of binary semaphores stored in
condition registers controlled by three instructions: inth.wait, inth.set and inth.clr. A
inth.wait checks whether a given condition is set. If it is, the condition is cleared;
otherwise the issuing thread is stalled until some other thread performs a inth.set
to that condition. If several inth.wait instructions accessing the same condition are
issued in parallel, only one of them will proceed. A inth.set sets the given condition.
If there was a inth.wait suspended on the same condition, the inth.wait is awakened
and the condition remains cleared. Finally, a inth.clr clears the given condition.

The programming model is described in detail in [30]. A similar architecture, although
using a dedicated namespace for shared registers, is described by Jesshope [44].

3.3.1 Communication Events in the Inthreads Model

A Synchronization event occurs between a inth.set and a inth.wait that accesses the
same condition. Synchronization events are relatively easy to detect, as the num-
ber of synchronization instructions that must be concurrently in flight is low (see
Section 3.5). inth.wait and inth.clr instructions never take a producer part in commu-
nication, and can be executed speculatively with no need for recovery.

A Variable value transfer event occurs between any instruction writing a value
to a variable (whether in a register or at some memory location) and a subsequent
instruction reading the value from that variable.

In contrast to the communication through synchronization, communication through
regular variables is harder to detect: any two instructions can potentially communi-
cate, and moreover, communication between memory instructions can be detected
only after the addresses for both instructions have been computed.

The Inthreads architecture handles the communication through variables at the
architecture level. To this end, the Inthreads-parallelized programs are required to
be data-race-free [2], i.e., to ensure that any two instructions that access the same
location are separated by synchronization. As a result, recovery of synchronization instructions implies correct squashing of all instructions involved in shared-variable communication.

A Thread starting event results from the communication between an \texttt{inth.start} and the first instruction of the started thread. Thread starting events are handled similarly to the synchronization ones. The only difference is that the instruction receiving the communication does not belong to a specific instruction type, but is just the first instruction started by the thread. For this, we hold a \texttt{tsv} for every thread, as if the whole thread receives the communication.

A Thread killing event occurs between an \texttt{inth.kill} and the first killed instruction of the target thread. While an \texttt{inth.kill} does not supply any “real” information, it does interact with the killed thread since the instructions following the \texttt{inth.kill} should not interact with the ones killed by it.

The situation is further complicated by the fact that \texttt{inth.kills} have a reverse effect on speculation: the instructions in the target thread are squashed only if the \texttt{inth.kill} is not misspeculated. Therefore, it is not sufficient to delay execution of an \texttt{inth.kill} until it becomes non-speculative, as instructions following it might interact with the target thread. Moreover, since an \texttt{inth.kill} terminates other instructions, recovery would be considerably more complex than just killing the dependent instructions. As a result, in this work we consider the \texttt{inth.kill} instructions as communication events, but execute them only non-speculatively.

In the rest of the paper, we denote the speculation settings with three letters, one for each speculation mechanism stated above. For example, TFF means that the speculative execution of synchronization instructions was turned on, and speculative value transfer between variables and speculative thread starting was turned off.

3.4 Implementation

In principle, the architecture of Inthreads is quite similar to SMT—both architectures execute multiple independent instruction streams on a shared execution core. Therefore, the microarchitecture re-uses most of the mechanisms present in SMT processors, such as multiple fetch units, multiple ROBs, shared physical register file and functional units and so on (a notable exception is the \textit{Register Allocation Table} (RAT), which is shared between threads in an Inthreads processor). Therefore, we take the SMT microarchitecture as a basis, and only describe the design of the thread management mechanisms.
The mechanisms that support Inthreads execution are outlined in Figure 3.4. The pipeline includes an additional stage, Instruction Wait, which implements delaying instructions of the threads which waiting on a condition. The delayed instructions are stored in the per-thread Wait Buffers (WBs). The WBs read the state of condition variables from the Available Conditions line to determine which instructions can be released.

The Thread Control Unit (TCU) orchestrates the execution of the threads on the processor. The Condition Speculation Table (CST) is used to keep track of the unresolved branch instructions. The computations performed by the TCU and the CST correspond to the multithreading speculation model developed in Section 3.2 and the Inthreads-specific details in Section 3.3.

The TCU, depicted in Figure 3.5, is used to 1) carry out the side effects of the synchronization and thread management instructions, 2) compute which instructions are speculative, and 3) control the squashing of dependent instructions in all the threads in response to a mispredicted branch.

The TCU holds the instructions in two queues, CIQ for the synchronization instructions and TMQ for the thread management instructions. Both CIQ and TMQ keep tsv of all the contained instructions. The CIQ and TMQ receive from the CST
the timestamps $S_E$ of the currently unresolved branches and use them to determine which of the instructions they contain are non-speculative and can be issued. In addition, the CIQ and TMQ receive information on branch mispredictions, in order to control the squashing of instructions from different threads.

Execution of synchronization instructions in the CIQ involves the computation of the currently available conditions, which are used by the WBs to determine which \texttt{inth.wait} instructions can be released. The \textit{Committed Conditions register} (CCR) holds the committed state of the condition variables. The value of the CCR is fed into the CIQ, which updates it according to the synchronization instructions (\texttt{inth.wait} and \texttt{inth.clr} instructions clean the corresponding bit, and \texttt{inth.set} instructions set it). When instructions are issued from the CIQ, they update the corresponding conditions in the CCR.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>No threads</th>
<th>Inthreads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline Length</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Supported threads</td>
<td>N/A</td>
<td>8</td>
</tr>
<tr>
<td>Fetch policy</td>
<td>8 per cycle</td>
<td>ICONT2.8</td>
</tr>
<tr>
<td>Branch predictor</td>
<td>bimodal</td>
<td></td>
</tr>
<tr>
<td>L1I size</td>
<td>64KB</td>
<td></td>
</tr>
<tr>
<td>Logical Registers</td>
<td>64GP+64FP</td>
<td></td>
</tr>
<tr>
<td>Physical Registers</td>
<td>512GP,512FP</td>
<td>384GP,384FP</td>
</tr>
<tr>
<td>ROB size</td>
<td>512</td>
<td>128*8</td>
</tr>
<tr>
<td>Issue Queue size</td>
<td></td>
<td>80</td>
</tr>
<tr>
<td>Memory Queue size</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>Functional units</td>
<td>6 Int,6FP,4 Branch</td>
<td></td>
</tr>
<tr>
<td>Max outstanding misses</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Max unresolved branches</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Max active synchronization instructions</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Memory ports</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>L1D size, latency</td>
<td>64KB, 1 cycle</td>
<td></td>
</tr>
<tr>
<td>L2 size, latency</td>
<td>1MB, 20 cycles</td>
<td></td>
</tr>
<tr>
<td>Memory latency</td>
<td></td>
<td>200</td>
</tr>
</tbody>
</table>

Table 3.1: Basic processor parameters

### 3.5 Evaluation

We have extended the SimpleScalar-PISA model [11] with the Inthreads-related extensions. The basic processor is modelled with a 8-stage pipeline, and the Inthreads-enabled variant has one more stage for synchronization functionality. Fetching was managed by the ICONT policy [84]. Table 3.1 summarizes the parameters.

We assume that the instructions dispatched to the TCU execute in as few as two cycles: one to compute the $tsv$, and one cycle to issue the instruction if it is ready. The logic involved in the TCU is similar to that of the Issue Queue, while the number
of instructions held in the TCU is significantly lower: in our experiments there was no measurable effect to increasing the TCU size over 16 instructions. Still, we measured the effect of increasing the TCU latency, shown in Figures 3.7 and 3.8.

The evaluation is based on three benchmarks from the SPEC2K suite [37]: 179.art, 181.mcf and 300.twolf, and four programs from the MediaBench suite [52]: Adpcm encode, Adpcm decode, G721 and Mpeg2. As a measure of the programs’ run time we used the number of clock cycles reported by the simulator.

3.5.1 Microbenchmark

To explore the performance behavior of the speculation mechanisms, we have implemented a microbenchmark with a tunable frequency of thread management events. The benchmark contains two nested loops. The inner loop consists of four independent sequences with heavy branching. The inner loop size, or the iteration size of the outer loop, determines the frequency of the synchronization, thread creation and termination events. The number of iterations of the outer loop determines the total number of such events.

The results are summarized in Figure 3.6. The first row of the graphs shows the speedup in comparison with the serial code, while the second one displays just the speedup that results from turning on speculation. The third row displays the percentage of squashed instructions.

The first three columns plot the performance for a fixed iteration size and varying number of iterations. The speedup remains relatively stable, except at a small number of iterations due to uneven branch prediction. Predictably, the speedup grows with the iteration size as the speedup caused by parallelization overcomes the thread management overhead. A more interesting effect is the growth in the additional speedup that results from speculation (row 2). Speculative execution of thread instructions allows different iterations of the outer loop to proceed in parallel, which is more important with a larger iteration size where the difference in the amount of work performed by the threads grows.

The rightmost three columns of Figure 3.6 show the behavior of parallelization for a given number of iterations of the outer loop. We can see that the speculation reaches its potential only with a relatively large number of iterations.

Finally, Figure 3.7 shows the effect of the latency of the TCU on performance. For the smallest possible parameters, a 14-cycle latency cancels out the benefits of parallelization. With larger parameters, the communication becomes less frequent, and the code is less sensitive to the latency.
### 3.5.2 SPEC2000 and Mediabench

The benefit of speculative execution depends on the amount of time saved by earlier execution of thread-related instructions and the frequency of such instructions. Table 3.2 shows the frequencies and average age of the `inth.set` and `inth.start` instructions,
Figure 3.8: Performance of SPEC and Mediabench benchmarks under varying TCU latency

measured from the time they enter the TCU and until they are issued.

The results of applying speculative execution of thread-related instructions to the benchmarks are summarized in Figure 3.9. The first row shows the overall speedup, and the second one—the speedup increment caused by speculation.

Mcf parallelizes consequent iterations that communicate heavily. This communication is sped up when executed speculatively, explaining the sensitivity of Mcf to speculative variable transfer. Art uses very fine grain synchronization, and thus receives most improvement from speculation on synchronization instructions. Both Art and Mcf perform thread starting relatively infrequently, and therefore do not benefit from speculation in thread starting.

In contrast, Twolf uses threads to parallelize small independent sections of code with heavy branching, and benefits from all the forms of speculative execution.

Both Adpcm programs split the execution into portions executed by threads arranged in a virtual pipeline, using synchronization at a low granularity. As a result, execution is sped up by speculative synchronization.

Both g721 and mpeg2 are barely affected by the speculation, albeit for opposite reasons. In g721, the hard-to-predict branches are executed just before synchronization, resulting in poor speculation success rate. In contrast, mpeg2 has long non-speculative sequences, obviating the need for speculative synchronization.

The effect of the TCU latency is shown in Figure 3.8. Mcf and Art are least sensitive due to the relatively long age of the synchronization instructions. The speedup of adding speculation to thread-related operations, shown in Figure 3.8b,
Figure 3.9: Performance of SPEC and Mediabench benchmarks with speculative execution

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>inth.set</th>
<th>inth.start</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Age</td>
<td>Freq</td>
</tr>
<tr>
<td>Mcf</td>
<td>41</td>
<td>0.018</td>
</tr>
<tr>
<td>Art</td>
<td>32</td>
<td>0.04</td>
</tr>
<tr>
<td>Twolf</td>
<td>4.4</td>
<td>0.057</td>
</tr>
<tr>
<td>Adpcm enc.</td>
<td>6.3</td>
<td>0.059</td>
</tr>
<tr>
<td>Adpcm dec.</td>
<td>2.4</td>
<td>0.061</td>
</tr>
<tr>
<td>G721</td>
<td>3.7</td>
<td>0.05</td>
</tr>
<tr>
<td>Mpeg2</td>
<td>1.5</td>
<td>0.014</td>
</tr>
</tbody>
</table>

Table 3.2: Average ages and frequencies of thread-related instructions
Figure 3.10: Performance of SPEC benchmarks under varying memory latency

decreases with the latency when it grows larger than the benefit of speculation. It is interesting to note that the speculation speedup for Mcf almost does not decrease with latency. The reason is that when the latency grows, additional parallelism is achieved by an increase in the number of iterations that execute in parallel.

Finally, for the SPEC benchmarks we have measured the effect of memory latency on speedup, shown in Figure 3.10. While the overall speedup may be sensitive to memory, the additional benefit of speculation remains almost constant.

3.6 Related Work

Synchronization operations are often redundant and can be ignored speculatively. Examples are Speculative Synchronization [59], Transactional Memory [38] and Speculative Lock Elision [66]. In contrast, Inthreads just speed synchronization up by executing it speculatively. This avoids the need to detect collisions, and also allows speculation in other thread operations, like starting and terminating.

Thread Level Speculation issues threads derived from a serial program, speculating on a fact that the threads will turn out to be independent. In Multiscalar processors [74], the program is partitioned into speculative tasks that execute concurrently while observing data dependencies. Speculative multithreaded processors [58] and Dynamic multithreading processors [3] generate threads at control speculation points.
In those works, the threads are usually very small, often with limited control flow. In contrast, the mechanisms in our work apply to general threads and provide a unified mechanism for speculation of both synchronization and thread management instructions. Other works in this area are [40, 45, 63, 64, 75, 76, 79].

The high cost of branch misprediction has prompted several works that aim at reducing the penalty by retaining those instructions that would execute in the same way regardless of the branch outcome [18, 68, 69]. Our approach can be seen as achieving about the same in software by speculatively starting threads with dependencies marked up by communication instructions.

3.7 Conclusion

This work provides a unified framework for speculative execution of multithreading-related instructions, including both synchronization primitives and instructions for starting and killing of threads. The framework can be applied to a given architecture by identifying the interactions between instructions of different threads.

The Inthreads architecture has four types of interactions: synchronization, data transfer through variables, thread starting and thread termination. All the types except for data transfer are linked to specific instructions and are easy to detect. The transfer through variables is handled by requiring the programs to be data-race free. As a result, misspeculation recovery of synchronization instructions implies correct recovery shared variable communication.

An additional aspect of this work is that our speculation mechanisms, due to the sharing of the processor resources among several threads, provide a good trade-off between the performance improvement and the increase in the percentage of squashed instructions. This implies that the techniques can be used for improving the power efficiency. We plan to explore this issue in depth in our future work.
Chapter 4

Using Fine Grain Multithreading for Energy Efficient Computing

Alex Gontmakher Avi Mendelson† Assaf Schuster
Technion, Israel Institute of Technology †Intel Labs, Haifa, Israel
{gsasha,assaf}@cs.technion.ac.il avi.mendelson@intel.com

Abstract

We investigate extremely fine-grain multithreading as a means for improving energy efficiency of single-task program execution. Our work is based on low-overhead threads executing an explicitly parallel program in a register-sharing context. The thread-based parallelism takes the place of instruction-level parallelism, allowing us to use simple and more energy-efficient in-order pipelines while retaining performance that is characteristic of classical out-of-order processors. Our evaluation shows that in energy terms, the parallelized code running over in-order pipelines can outperform both plain in-order and out-of-order processors.

4.1 Introduction

Until recently, processors have been designed with a single goal in mind — that of high performance. Most of the techniques for achieving high performance are based on

1Published in 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP07) [26].
Two leading paradigms for parallelism exploitation are out-of-order execution, OoO (as opposed to in-order execution, InO) that takes advantage of independence of consequent instructions, and multithreading that supports execution of explicitly parallel, independent code streams. The run-time nature of OoO parallelism incurs significant complexity and power overhead. Indeed, as Figure 4.1 shows, OoO processors are used only when high single-task performance is mandatory. For cases where only the throughput is important, processors usually include support for thread-level parallelism in the form of SMT or CMP, and energy efficiency is achieved by employing an InO pipeline. An extreme example of this approach is SUN’s Niagara processor [94], which runs up to 32 threads on a single chip.

To visualize the impact of the choice between OoO and InO pipeline designs on processor complexity and energy efficiency, consider the following processor pairs: MIPS R5K vs. MIPS R10K [90, 91] and Alpha 21164 vs. Alpha 21264 [88, 89]. Processors in each pair implement the same ISA at a similar technology level and differ mainly in the choice between InO or OoO design\(^2\). As Figure 4.2 shows, transition to OoO resulted in about twofold performance improvement at the cost of three times as

\(^2\)These models are representative of a switch from InO to OoO and are therefore closely related.

More recent processors are designed either for low power or for high performance from the bottom up and are related much more loosely.
much power. Normalizing to performance, we observe power efficiency loss of $1.63 \times$ for MIPS R10K vs. R5K, and of $1.81 \times$ for 21264 vs. 21164.

We propose to bring the energy benefits of thread-based parallelism to single task execution. To this end, we increase performance through fine granularity multithreading and improve energy efficiency by employing an in-order pipeline. Essentially, this replaces dynamic parallelism discovery of OoO with explicit parallelism specified at compile time.

To implement fine granularity multithreading, we use the Inthreads programming model. The model is based on a fixed number of register-sharing threads that execute within the context of a single conventional SMT thread. The threads share the registers cooperatively, using disjoint register subsets to avoid conflicts. In addition, the architecture supports communication through the shared registers with a special set of synchronization instructions. The fixed nature of the Inthreads architecture enables completely hardware-based implementation, resulting in an extremely low overhead of thread management and synchronization. This allows us to parallelize portions of code containing as few as several dozen instructions. Inthreads parallelization can be applied to programs at different levels of abstraction and automatization. The results described in this paper are based on a C-derived intermediate language that includes direct representation of the parallelism. The programming model and parallelization approaches are described in Section 4.2.

<table>
<thead>
<tr>
<th></th>
<th>MIPS R5K</th>
<th>MIPS R10K</th>
<th>Alpha 21164</th>
<th>Alpha 21264</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline</td>
<td>InO</td>
<td>OoO</td>
<td>InO</td>
<td>OoO</td>
</tr>
<tr>
<td>SPECInt95</td>
<td>5.5</td>
<td>10.1</td>
<td>15.4</td>
<td>27.7</td>
</tr>
<tr>
<td>SPECFP95</td>
<td>5.5</td>
<td>9.71</td>
<td>21.1</td>
<td>58.7</td>
</tr>
<tr>
<td>Power (W)</td>
<td>10</td>
<td>30</td>
<td>28</td>
<td>91</td>
</tr>
<tr>
<td>Clock Rate</td>
<td>200</td>
<td>250</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>Technology ($\mu$)</td>
<td>0.35</td>
<td>0.35</td>
<td>0.35</td>
<td>0.35</td>
</tr>
<tr>
<td>Transistors</td>
<td>3.6M</td>
<td>6.7M</td>
<td>9.3M</td>
<td>15.2M</td>
</tr>
<tr>
<td>Transistors (Logic)</td>
<td>0.8M</td>
<td>2.3M</td>
<td>2.6M</td>
<td>6M</td>
</tr>
<tr>
<td>Physical Registers</td>
<td>N/A</td>
<td>64</td>
<td>N/A</td>
<td>80</td>
</tr>
</tbody>
</table>

Figure 4.2: Energy and efficiency comparison of InO and OoO processors
In the rest of this paper, we use OoO to denote the classic model of an out-of-order processor running a sequential program, InO to denote a classic in-order processor running a sequential program, and InT for the model we introduce in this work, an in-order processor running a program parallelized with Inthreads.

The contributions of this paper are threefold:

1. We show that a synchronization mechanism combined with compile-time consistency support reduces complexity of the hardware that deals with consistency of concurrent accesses to shared locations.

2. We compare the speedup potential of fine granularity parallelization to the speedup that results from instruction-level parallelism discovered by OoO execution. Our evaluation shows that execution speed of programs parallelized with Inthreads on an InO processor is on par with the speed of the original code on an OoO processor.

3. We compare the energy efficiency of fine-grain multithreaded code running on an InO processor to the efficiency of the original code on an OoO processor. Our results show that the InT model improves the efficiency compared to OoO under both Energy ($E$) and Energy-Delay Product ($ED$) metrics [32].

4.2 Programming Model

4.2.1 Inthreads Instruction Set Architecture

The Inthreads programming model is based on a fixed number of threads running over shared architectural registers in the context of a single conventional SMT thread. The model provides an extremely lightweight threading mechanism since the fixed number of threads allows for thread management to be performed entirely in hardware. In addition, the shared registers are used for communication: a value can be transferred between threads by writing it into a register in one thread and reading it from the same register in another.

The Inthreads Instruction Set Architecture (ISA) provides three new instructions for starting and stopping threads: `inth.start`, `inth.halt` and `inth.kill`. `inth.start` creates a new thread with a given thread ID at a given address. A thread can request to terminate itself with `inth.halt` or any other thread using `inth.kill`. `inth.halt` is executed synchronously, guaranteeing completion of all the preceding instructions.
To support communication through registers, the ISA includes a synchronization mechanism. The mechanism consists of a set of binary semaphores stored in *condition registers* and manipulated by three instructions: `inth.wait`, `inth.set` and `inth.clr`. A `inth.wait` checks whether a given condition is set. If it is, the condition is cleared; otherwise, the issuing thread is stalled. A `inth.set` releases a `inth.wait` suspended on the same condition, or sets the condition register if no `inth.wait` was suspended. Finally, a `inth.clr` clears the given condition.

The synchronization mechanism is based on collaboration between the software and the hardware. Specifically, the hardware provides *Data-Race-Free-1 (DRF1)* \[2\] consistency model for register and memory accesses with respect to the thread-related operations, and the software is required to be free of data races \[2, 25\]. In addition, the compiler supports the collaboration by preserving the consistency properties in the generated code. The software and hardware implementation of the DRF1 support is discussed in more detail in Sections 4.2.3 and 4.3.3.

### 4.2.2 Parallelization

The Inthreads compilation flow proceeds in two stages, shown in Figure 4.3: the *parallelization* and the *code generation* for parallelized code. As an interface between the two stages, we defined *Inth-C*, an extension of the C language that adds explicit parallelism constructs. Inth-C defines constructs for denoting explicitly parallel sections of code and a full set of commands that correspond to the Inthreads ISA instructions. Furthermore, the Inth-C compiler recognizes the commands’ execution semantics and supports communication between threads by detecting the shared variables and guid-
ing the code optimizations appropriately. The compiler implementation is discussed in more detail in Section 4.2.3.

The parallelization can be obtained in several ways, with different levels of automation. The lowest level option is manual embedding of fragments of Inth-C parallelized code directly into the host program. Inth-C programming is rather straightforward, as shown below.

For a more high-level parallelization strategy, we envision using existing parallel language extensions such as OpenMP [17]. This would allow the programmer to specify the parallelism declaratively, abstracting from the low level details of parallel code execution. Most of the basic OpenMP declarations, such as parallel for and parallel sections, can be supported directly by Inthreads. The Inthreads model would impose several limitations on the OpenMP parallelization: for example, the whole sections of parallelized code must be lexically visible. The limitations are balanced by the lower overhead of Inthreads, which allows finer-granularity parallelization.

With this strategy, Inthreads and conventional multithreading would occupy different niches of parallelization granularity: the thread-based implementation can be used for long-running sections of code, while the Inthreads-based one would be better for short sections. Section 4.4.2 shows the overhead gap between Inthreads and thread-based parallelization, implying that such a distinction is desirable. In effect, Inthreads would extend the reach of OpenMP to small portions of code that could not be profitably parallelized with traditional threads.

Figure 4.21 presents an example of code parallelized with Inth-C and OpenMP. The loop in the example performs several tens of iterations, with the loop body taking less than 100 cycles on average. It is clearly not large enough to be parallelized with conventional threads, but still can be optimized using Inthreads. The main difference between Inth-C and OpenMP parallelization is that Inth-C requires the code to be manually split into threads, using explicit thread-management and synchronization. In contrast, the OpenMP parallelization can be done by just adding two compiler directives (to extract all the available parallelism, the variable next had to be privatized).

Finally, automatic compile-time parallelization can be achieved using techniques available in existing compilers [56]. The problems in static parallelism discovery, such as the need for accurate analysis of aliasing and control-flow dependencies, will be less pressing in Inthreads than in SMT parallelization. A more local parallelization context will reduce the amount of code which must be analyzed for collisions and the chance of such collisions to occur. In addition, the low parallelization overhead allows parallelization of smaller portions of code, which have greater chance to contain
Figure 4.4: Parallel code that can be miscompiled by a non-threads-aware compiler

-independent computation. Moreover, Inthreads parallelization can be used for code vectorization [4], and can support more complex cases including control flow.

The results presented in this paper are based on manual parallelization of C code into Inth-C.

### 4.2.3 Code Generation

In this section we discuss the extensions that were necessary for correct compilation of Inth-C programs. We have implemented the Inth-C compiler as an extension to GCC [93]. To support Inth-C, we extended GCC’s parser with additional block attributes corresponding to the \#pragma inthread and Inthreads-specific operations such as INTH_START, INTH_SET and INTH_WAIT.

The Inth-C compiler optimizations are required to preserve the data-race-free properties of the program and the DRF1 model visible by the code, demanding adjustments at all the levels of the compiler backend, from the intermediate representation to the optimization algorithms. For an example of issues that need to be addressed, consider the code in Figure 4.4. Variable \( a \) is assigned in thread \( T_1 \) and is later read in \( T_0 \). The order of accesses to \( a \) is ensured by synchronization on condition register 1. However, a traditional implementation of dead code removal optimization would not be aware of the communication through \( a \). Seeing no uses of \( a \) in \( T_1 \), it would consider the assignment to \( a \) in \( T_1 \) as dead, ultimately removing the assignment statement and thus generating incorrect code.

Another issue the compiler needs to handle in compilation of Inth-C is register allocation. Consider again the communication through \( a \) in Figure 4.4. For the
communication to proceed correctly, \( a \) must be allocated to the same location (register or stack entry) in both threads. In addition, the compiler must make sure that \( b \) and \( c \) use different registers, otherwise a data race could occur.

To adapt the compiler’s internal representation to Inthreads compilation, we have extended the control flow graph (CFG) with parallelism information, forming a Concurrent CFG (CCFG) [53, 77, 78]. Our variant of CCFG contains two additional edge types. Synchronization edges connect \texttt{INTH\_SET} instructions to the corresponding \texttt{INTH\_WAIT} ones. Parallel edges model thread starting events. The parallel edges are used to compute the set of instructions that participate in a parallelized region.

Figure 4.5 shows the CCFG graph for the program in Figure 4.4. The CCFG includes a synchronization edge (dashed) from \texttt{INTH\_SET}(1) to \texttt{INTH\_WAIT}(1) and two parallel edges (bold) outgoing from \texttt{INTH\_START}. Also note that there are no outgoing edges from \texttt{INTH\_HALT}.

The additional information in CCFG is used by a variety of analysis passes related to parallel code compilation. Prior to the optimizations, the compiler uses the synchronization and parallel nodes to determine the set of shared variables (those that are accessed for reading and writing concurrently in different threads and can therefore be used for communication). The rest of the variables, including those that are only read in concurrent threads, are treated as thread-private.

The shared variables are handled differently from the thread-private ones. Since they are used for communication, they need to obey the ordering semantics with
respect to the synchronization instructions. Moreover, the register allocation for shared variables must be performed consistently in order to support communication through them. The first requirement is ensured by explicitly maintaining the order between the shared variable accesses and the synchronization instructions. The second requirement follows naturally from including the concurrent edges into the data flow analysis: the conflicting accesses are noted and standard coloring register allocator will be able to handle them with little modification.

Given the CCFG, most data flow-dependent optimizations can be easily adapted for Inth-C code. The data flow independent optimizations, such as branch elimination, do not need modifications for Inth-C parallelism.

Consider the application of dead code elimination to the code in Figure 4.4 using the CCFG shown in Figure 4.5. The algorithm would start by noting the return a statement as essential (i.e., not dead), and then, following the flow graph, would recognize the assignment a+=b as essential as well. Now, following the synchronization edge from INTH_SET(1) to INTH_WAIT(1), it will recognize that the assignment to a in T1 is eventually used, correctly recognizing the statement as essential.

For further information on Inth-C compilation, see [73].

4.3 Microarchitecture

Figure 4.6 shows the differences between the pipelines of InO, OoO and InT processors. The OoO and InO pipelines differ significantly, except for the front-end stages. The InT pipeline is based on the InO one, with several extensions similar to those used in SMT processors [84]. First, the Fetch unit is extended with multiple PCs and buffers, capable of fetching instructions of several threads per cycle. Second, a new stage, Instruction Wait, takes care of delaying the inth.wait instructions. Similarly, we extend the issuing capability to process instructions of different threads independently.

In the rest of this section we discuss the details and complexity of the Inthreads-related mechanisms.

---

3 An implementation of Inthreads based on an out-of-order pipeline is possible, but would take a different position in the power/performance tradeoff. It would improve performance of single-task execution but would offer only limited improvement in energy efficiency.
Figure 4.6: Pipeline differences for the three processor models. The shaded items are Inthreads-related.

4.3.1 Fetch Stage

Figure 4.8 shows the difference between single-threaded and multithreaded fetching. Each thread has a private buffer for instructions brought from ICache. The buffers access the cache autonomously, arbitrating by round-robin.

While the processor needs to feed instructions to several concurrent threads, allowing all of them to fetch at the same time is prohibitively complex and unnecessary. It is sufficient to allow only a small number of threads to fetch concurrently, arbitrating between them to ensure fairness. The algorithm for selecting the threads to fetch is much simpler than the ICOUNT policy [85] of SMT: we just disable fetching for threads that are stuck on a inth.wait instruction or on issuing. The reason that we can use a simpler algorithm is that unlike SMT, an InO processor does not have a shared issue queue, and therefore one thread cannot starve another.

To evaluate the complexity of InT fetching, we explored different combinations of the number of cache ports $P$ and the number of threads that fetch in a cycle $F$. The results, shown in Figure 4.7, indicated a diminishing benefit in having $P > 2$ and
Figure 4.7: Execution time as a function of the fetch policy. $P$ is the number of Icache read ports, $T$ is the number of threads that can fetch per cycle.

$F > 2$. Consequently, the complexity added to the fetch logic is minimal: we either have only one active buffer or mix instructions from two buffers by allowing each to supply half of the fetched instructions. The inactive PCs and buffers are disabled with clock gating to further conserve energy.

The decoding process needs no prioritization mechanisms and is not affected by the fact that the instructions belong to different threads.

### 4.3.2 Instruction Wait Stage

The purpose of the *Instruction Wait* stage is to support the synchronization performed by `in\text{th}.\text{wait}` and `in\text{th}.\text{set}` instructions. For each `in\text{th}.\text{wait}`, we check whether the condition it requires is available and stall the thread if necessary. For each delayed `in\text{th}.\text{wait}` instruction (naturally there can be only one in each thread), the Instruction Wait stage requests the Fetch stage to pause delivering instructions for the corresponding thread.

The Instruction Wait stage keeps for each thread the number of the condition register that the thread depends upon, and a $32\times1$ bit table holding the current status of the condition registers. When a `in\text{th}.\text{set}` is executed (after being released from Issue), its condition number is compared to the condition register numbers kept for the stalled threads. If a match is found, the corresponding thread is released and allowed to fetch instructions again. If no thread required the condition, the corresponding condition register is set.

Our evaluation has shown that synchronization instructions occur relatively rarely, on the order of 2% of dynamic instructions. As a result, the Instruction Wait stage has little activity and low energy consumption.

The execution of `in\text{th}.\text{set}` and `in\text{th}.\text{clr}` instructions involves sending the affected
condition number to the Thread Waiting stage, where the change is reflected in the 32x1 bit Conditions table. In addition, as shown in Figure 4.13, there is relatively little activity in the condition registers, implying very little energy consumption of the logic.

### 4.3.3 Instruction Issue

An OoO processor contains a large, complex Issue Queue (IQ) for dependency checking. The corresponding structure of an InO processor is significantly simpler since
only instructions at the head of the queue can be issued. As a result, few instructions need to be checked for dependencies, and moreover, there is no reason to make the queue as large as the OoO one.

While an InT processor issues instructions from different threads concurrently, its dependency checking is not much more complex than that of InO: since the program is required to be data race free, *only instructions issued by the same thread need to be checked for dependencies*.

To prove this claim, we consider two instructions $i_1, i_2$ belonging to two threads and accessing the same resource. The accesses of $i_1$ and $i_2$ constitute a potential data race and must be ordered by synchronization. We assume WLOG that $i_1$ is ordered before $i_2$.

We will show that $i_1$ will have started executing by the time $i_2$ arrives to Issue. This is the same effect that we would expect for issuing of two dependent instructions of the same thread. Consequently, the existing single-threaded execution logic will ensure correct results for the interaction between $i_1$ and $i_2$.

The synchronization sequence is a chain of instructions of the following form: $i_1 \downarrow [s_{rel} \searrow s_{acq} \downarrow]^+i_2$, where $i \downarrow j$ denotes that $i$ precedes $j$ in the program order of the same thread and $i_{rel} \searrow j_{acq}$ denotes a pair of interacting synchronization instructions, where $i_{rel}$ has release semantics and $j_{acq}$ has acquire semantics.

Since $i \downarrow j$ belong to the same thread, they are submitted for issue in the program order, and, because of in-order issue, $i$ is necessarily issued before $j$. For $i_{rel} \searrow j_{acq}$, we distinguish two types of synchronization sequences:
1. $i_{rel}$ is a \texttt{inth.set} and $j_{acq}$ is a \texttt{inth.wait}. Our implementation, described in Section 4.3.2, delays \texttt{inth.wait} in the Instruction Wait stage, prior to issuing. The \texttt{inth.set} wakes up the \texttt{inth.wait} during execution.

2. $i_{rel}$ is an \texttt{inth.start} and $j_{acq}$ is an instruction of the started thread. Since the execution of $i_{rel}$ introduces $j_{acq}$ into the processor, $j_{acq}$ is fetched only after $i_{rel}$ executes, and consequently, $j_{acq}$ is available for issue only after $i_{rel}$ has been executed. Q.E.D.

Taking advantage of the lack of dependence between instructions from different threads, we duplicated the issuing logic of InO for each supported thread. Similarly to InO, only few instructions can be issued per cycle: our evaluation, shown in Figure 4.10, indicates that the optimal value is 4. Since we support a maximum of 8 threads, there is a total of 32 IQ entries per processor. However, most of our parallelization was done with 2 to 4 threads. The IQs of the other threads are idle most of the time and take little energy thanks to clock gating.

We have considered the alternative of mixing instructions of all the threads in a single RS. However, this structure is unable to allow one thread to proceed while another slows down, and performed only slightly faster than a single-threaded Inorder processor.

4.4 Evaluation

4.4.1 Simulation

Our simulation framework is based on the SimpleScalar simulator [11] running the PISA instruction set. We evaluate the energy consumption using Wattch [10]. We extended Wattch to model in-order execution by adjusting the relevant processor.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>OoO</th>
<th>In-Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline Length</td>
<td>12</td>
<td>8 (9)</td>
</tr>
<tr>
<td>Max threads</td>
<td>N/A</td>
<td>8</td>
</tr>
<tr>
<td>Fetch width</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Branch predictor</td>
<td>bimodal</td>
<td></td>
</tr>
<tr>
<td>L1I size</td>
<td>64KB</td>
<td></td>
</tr>
<tr>
<td>Physical Registers</td>
<td>128GP,128FP</td>
<td>N/A</td>
</tr>
<tr>
<td>ROB size</td>
<td>128</td>
<td>N/A</td>
</tr>
<tr>
<td>Issue Q size</td>
<td>40</td>
<td>N/A</td>
</tr>
<tr>
<td>Memory Q size</td>
<td>20</td>
<td>N/A</td>
</tr>
<tr>
<td>Functional units</td>
<td>4 Int, 4FP</td>
<td></td>
</tr>
<tr>
<td>L1D size</td>
<td>64KB</td>
<td></td>
</tr>
<tr>
<td>L2 size</td>
<td>1MB</td>
<td></td>
</tr>
<tr>
<td>L2 latency</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Memory latency</td>
<td>200</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.11: Baseline processor parameters

structures according to the changes described in Section 4.3. We use the CC3 clock gating style which models the static energy leakage.

The processors of all three models were configured with the same amount of execution resources, such as caches and functional units. Figure 4.11 lists the configuration parameters of the models.

We evaluate our approach using three benchmarks from the SPEC2000 benchmark suite [37]: Art, Mcf and Twolf, and most of the Mediabench benchmarks [52]. The parallelization achieved multithreaded execution for the majority of dynamic instructions. All the benchmarks were executed until completion, with inputs that required up to a billion cycles to execute.
4.4.2 Inthreads Overhead Characterization

Before discussing the speed and energy results of Inthreads execution, we provide an intuition for the overheads incurred by its operation. To this end, we built a microbenchmark that runs two tasks in parallel. Each task is a loop with a body of about 10 instructions. We estimate the overhead by varying the number of iterations and measuring the speedup over serial execution. When the speedup is close to one, the parallelization benefit just barely overcomes the overhead. When the loop is large enough for the overhead to become insignificant, we expect to reach the maximal speedup of $2 \times$.

The Inthreads parallelization proceeds by starting an additional thread to split the work. For SMT parallelization, two approaches were attempted: starting a thread ev-
every time and keeping a background thread that receives the assigned jobs by synchronization. The program, parallelized with pthreads was executed on a dual PowerPC running Linux 2.6. The results, shown in Figure 4.14, indicate that the Inthreads overhead is about three orders of magnitude lighter than pthreads one.

As a result of the low overhead, we are able to use Inthreads at an extremely fine granularity. To judge the granularity of the actual parallelization, we measured the average number of instructions executed in each parallelized and non-parallelized region of code of the benchmarks we used. The results, shown in Figure 4.1, indicate that on average, there are several hundred instructions between thread splits and thread joins, well below the limits practical for conventional threads.

### 4.4.3 Performance

Programs differ widely in their ability to benefit from OoO and InT execution. Figure 4.12 shows the execution time for our benchmarks under InO and InT models, normalized to that of OoO. We can see that while InO performs significantly worse than OoO, in most cases InT closely approaches the speed of OoO (except for Art, where InO is at a huge disadvantage to begin with), and occasionally exceeds it. On average, both OoO and InT exhibit similar performance, 2x faster than InO.

Inthreads parallelization increases the number of instructions executed by the program as a result of the thread manipulation code. To gauge the overhead caused by the additional instructions, we computed dynamic instruction counts of the serial and the parallelized code, shown in Figure 4.13. On average, the number of executed
instructions increases by less than 5%, about half of which is attributed to the thread manipulation and synchronization instructions. Another observation from Figure 4.13 is that InT execution, similarly to InO, fetches 20% less instructions than OoO, implying less wasted work and another source for improved efficiency.

### 4.4.4 Energy Consumption

Several key differences in the pipeline organization contribute to InO’s improved energy consumption. First, an OoO processor requires many activities absent from an InO one. These activities include Register Renaming and Reorder Buffer (ROB) management. In addition, execution of some instructions is inherently more complex in OoO. For instance, OoO processors split the memory accesses into the address computation and the memory access instructions to facilitate quicker aliasing resolution.

Second, the out-of-order issuing and scheduling logic of OoO is more complex and the sizes of the structures implementing that logic are inherently larger, as shown in Section 4.3.3. In addition, in an in-order processor, all the register accesses are served from the architectural register file, which is smaller than the physical one of an out-of-order processor. Finally, the simpler logic of the in-order pipeline results in a smaller chip, reducing the clock distribution and wakeup signalling energy.

<table>
<thead>
<tr>
<th></th>
<th>Parallel regions</th>
<th>Serial regions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adpcm</td>
<td>227</td>
<td>4.3</td>
</tr>
<tr>
<td>G721</td>
<td>131</td>
<td>102</td>
</tr>
<tr>
<td>Mpeg2</td>
<td>474</td>
<td>119</td>
</tr>
<tr>
<td>Epic</td>
<td>610</td>
<td>95</td>
</tr>
<tr>
<td>Gsm</td>
<td>1197</td>
<td>172</td>
</tr>
<tr>
<td>Mesa</td>
<td>2716</td>
<td>133</td>
</tr>
<tr>
<td>Mcf</td>
<td>890</td>
<td>719</td>
</tr>
<tr>
<td>Art</td>
<td>1767</td>
<td>280</td>
</tr>
<tr>
<td>Twolf</td>
<td>100</td>
<td>91</td>
</tr>
</tbody>
</table>

Table 4.1: Parallelized region sizes
Figure 4.15 shows the energy and ED results of our benchmarks. For most of the programs, the energy difference between InO and InT is small and results mostly from the leakage because of the increased running time. Most benchmarks consume about 50% less energy per task under InO than under OoO. The only exception is Art, which is significantly less energy efficient under InO — its energy consumption is dominated by leakage power due to its very low utilization of processor resources.

The results change dramatically when the execution time is brought into consideration. With the ED metric, three of the benchmarks perform significantly worse under InO than under OoO, while under InT, all of the benchmarks exhibit better results than under OoO. On average, the difference in ED between InO and InT is more than $2\times$. As a result, while OoO is preferred to InO under ED, InT is significantly better than either.

A more detailed view of the performance can be achieved by calculating the energy that each one of the models consumes to achieve a certain level of IPC. To this end, we divide the execution of the benchmarks into portions of 1000 cycles and plot the energy consumption during the portion versus the achieved IPC. The result is shown in Figure 4.16. We can see that an InT processor expends about the same amount of energy to reach the same IPC as InO, and for higher IPCs, the same energy that InO would spend if it could reach that performance.

It is instructive to check the scaling capacity of different processor models, as
scaling can potentially improve performance at the price of increased energy consumption. We have executed the same measurement on a processor which is twice as wide, able to fetch 8 instructions per cycle. As expected, an InO processor does not benefit from the scaling at all, and therefore, will expend more energy to compute the same task. However, both InT and OoO are able to scale, with InT processor being still better than OoO in energy efficiency.

Figure 4.18 shows the breakdown of the energy consumption, grouped into the main processor components: Fetch — energy of the ICache and the fetch logic, Datapath — energy consumption related to the instruction execution, Control — energy expended by the various bookkeeping tasks, Memory — energy of the data-related memory accesses, and Clock. The most notable improvement occurs in the control logic of the processor, which is significantly simpler in the case of an in-order pipeline. The improvement of the datapath energy consumption results from accessing the architectural, rather than physical, register file. Moreover, in an InT pipeline, fewer ALU operations and register accesses are squashed prior to completion. Finally, the InT improves the clock distribution energy in comparison to InO due to its shorter execution time.

The last measurements in this section are related to the resource configuration parameters described in Sections 4.3.1 and 4.3.3. Figure 4.19 presents energy con-
Figure 4.17: Processor energy consumption as a function of the IPC at fetch width 8

Figure 4.18: Breakdown of energy consumption to processor components
sumption for different variants of the fetch policy. We can see that the chosen configuration of 2 Icache ports and 2 threads improves the energy by 2.5% on average and $ED$ by 7%. Figure 4.20 shows energy consumption for different RS sizes. Due to increased complexity, growing RS beyond 4 entries results in an overall increase in the energy consumption.

4.5 Related Work

The role of multithreading in the trade-off between performance and efficiency has received a lot of attention. [39] suggests that for a throughput-oriented workflow, an in-order SMT processor makes as much sense as an out-of-order SMT one. While the paper does not state this directly, the reduced complexity should improve the processor’s energy efficiency. Indeed, in-order SMT architectures have recently been implemented commercially [94].

Even under out-of-order execution, a SMT processor is more energy efficient than a single-threaded one. [71] shows 22% improvement in energy per instruction for 4 threads, and shows that such processor can trade complexity for performance while still being faster than a single-threaded processor. Li et. al [55] perform an extensive examination of the power-performance design space. Alternatively, a CMP processor, or a combination of CMP and SMT can be used to construct an efficient processor [70, 54].

Alternatively, speeding up an in-order processor to the speed of an out-of-order one can result in significant energy savings even without multithreading. Barnes et. al. [7] achieve this through multipass execution, which can speculatively execute instructions following a missing load and later reuse the results of instructions that have been executed correctly.

VLIW architectures expose instruction-level parallelism to the processor by marking up independent instructions directly in the instruction set. Such architectures rely on the compiler to discover the parallelism, and therefore can achieve highly energy-efficient execution. A prominent example of this approach is the Transmeta processor [92] which translated x86 instructions to internal VLIW machine code in order to reduce power consumption.

The above benefits in energy efficiency may be restricted to multithreaded workloads, where only throughput is important. However, if single task performance is necessary, the same energy benefits can be realized through fine granularity parallelization. [43, 44] describe a Microthreaded architecture that, similarly to Inthreads,
partitions a single context into multiple threads that interact by communicating through registers. The authors mention the potential for energy efficient execution, but no detailed evaluation was presented. The main difference between Microthreading and Inthreads is that our model aims at a fixed parallelization level. The resulting simplification of the processor allows for better energy efficiency.

It was shown that SMT can also take advantage of efficient hardware-based synchronization primitives [86]. The main difference of our synchronization mechanism is its reliance on software consistency guarantees, which allows for a simpler and more efficient implementation.

Processor resources are often underutilized, and therefore, significant energy can be saved by turning off parts of the processor when they are unused or are unable to contribute to the performance. The issue queue can be partially deactivated [12, 23] or simplified with little cost in performance [13]. Register file energy can be reduced using banking [16, 83] or hierarchical organization [6].

Another source of inefficiency of out-of-order is the excessive number of speculative instructions entering the processor. The problem can be mitigated by throttling the pipeline when low-confidence branches are encountered [57]. The throttling is even more efficient in SMT processors, where another thread can utilize the freed fetch bandwidth for useful computation.

4.6 Conclusions

Explicit thread-based parallelism can be useful in many ways, from enhancing program scalability to improving the throughput of certain workloads. For tasks that are only throughput-limited, one can achieve significant energy savings by giving up on processor complexity and relying on multiple simple and energy efficient processors instead of few powerful ones. This work takes a similar approach to single task computing. By utilizing Inthreads, an extremely fine-granularity architecture, we are able to replace the costly dynamic parallelism discovery logic of out-of-order execution with compilation-time explicit parallelization. Running the explicitly parallel program over an in-order pipeline results in a significantly reduced energy consumption while keeping the same level of program performance.

At low IPC levels, the InT model consumes about the same energy as InO. Furthermore, at higher IPC levels, our model expends the same energy that InO would need if it could reach that level of IPC. We can therefore consider Inthreads as a method of feeding an in-order pipeline with an improved instruction stream with fewer de-
dependencies and better available parallelism. The efficiency is further improved by the fact that the synchronization mechanism, which relies on software consistency model, guarantees lack of dependencies between instructions of different threads, reducing the amount of runtime checking necessary.

In this work, we tried to reduce the overhead of Inthreads-related logic to the maximal extent possible. However, there are certain mechanisms, such as speculative execution of synchronization and thread management, that improve performance but can have a significant impact on energy consumption. In the future work, we plan to carry out careful analysis of these mechanisms to determine whether they are beneficial from the energy efficiency point of view. In addition, it will be interesting to evaluate the effect of Inthreads on energy efficiency under an OoO pipeline.
Figure 4.19: Execution time as a function of the fetch policy. $P$ is the number of Icache read ports, $T$ is the number of threads that can fetch per cycle.

Figure 4.20: Energy and $ED$ as a function of the RS size
next = 0;
for (i=2; i<=N;i++) {
a = perm[i]->a;
r = compute_r(a);
if (cond(a,r)) {
    next++;
    perm[next]->a = a;
    perm[next]->cost = r;
}
}

#pragma omp parallel
private(a,r,next_p)
for (i=2; i<N; i++) {
a = perm[i]->a;
r = compute_r(a);
if (cond(a,r)) {
    #pragma omp ordered
    next_p=next++;
    perm[next_p]->a=a;
    perm[next_p]->cost=r;
}
}

worker1:
#pragma inthread
{
    int i;
double a, r;
for(i=2;i<=N;i+=2)
{
a = perm[i]->a;
r = compute_r(a);
res1 = cond(a,r);
INTH_SET(1);
INTH_WAIT(2);
if (res1) {
    d1 = ++next;
    INTH_SET(2);
    i++;
    /*Interact with wkr 2*/
    ...
}
INTH_WAIT(1);
/* wait for workers to complete */
INTH_WAIT(1);
INTH_HALT();
}

INTH_START(1, worker1);
INTH_START(2, worker2);
#pragma inthread
{
for (i=2, next=0; ;) {
    if (!i <= N) break;
    /*Interact with wkr 1*/
    INTH_WAIT(1);
    if (res1) {
        d1 = ++next;
        INTH_SET(2);
        i++;
        /*Interact with wkr 2*/
        ...
    }
    /* wait for workers to complete */
    INTH_WAIT(1);
    ...
}

Coordinating thread
Worker thread

b) Inth-C parallelization

Figure 4.21: Inthreads and OpenMP parallelization example. The original code is part of Spec2K Mcf benchmark. The computation of comp_r and cond is independent in all the loop iterations, and the only operation that must be performed serially is the update of next.
Chapter 5

Correctness Aspects of a Register Sharing Architecture

Alex Gontmakher † Avi Mendelson Assaf Schuster
Technion, Israel Institute of Technology † Intel Labs, Haifa, Israel
{gsasha@cs,assaf@cs.technion.ac.il} avi.mendelson@intel.com

Gregory Shklover Vladimir Zdornov
Technion, Israel Institute of Technology
{gkovriga@cs,zdv@tx.technion.ac.il}

Abstract

The Inthreads architecture enables fine-grain parallelization by tight integration between the software and the hardware. The architecture is based on a programming model which specifies allowed software behaviors and the processor’s execution guarantees. The model benefits both the software, by simplifying compiler optimization and the hardware, by reducing the dependencies between instructions of the different threads.

In this paper we study the correctness of all the aspects of the Inthreads architecture. First, we present a programming model for register-shared multithreading. Using the model, we determine correctness requirements for the compiler and the software. We then apply these requirements to the analysis of the compiler optimizations and the

1Submitted for publication. Portions accepted for publication at the 2007 International Conference on Parallel Processing (ICPP’07).
5.1 Introduction

Modern architectures realize most of the parallelism existing at the finest instruction-level granularity (ILP). Consequently, recent microprocessors increasingly support coarser parallelism in the form of Simultaneous Multithreading (SMT) [84] and Chip Multiprocessing (CMP) [35].

The Inthreads architecture targets medium-to-low parallelism, between that of ILP and that of multithreaded execution. To this end, it applies thread-based parallelization at a fine granularity using extremely lightweight threads. To reduce the overhead as much as possible, Inthreads adopts a programming model which shares the contexts of the threads to the maximal possible extent, including not only the memory address space, but also most of the architectural registers. The model is based on a collaboration between the software and the hardware. The hardware provides a set of synchronization instructions and ensures their ordering semantics; the responsibility of the software is to utilize the semantics to ensure correct and consistent program execution.

The programming model provides a formal specification of the correctness requirements from a processor and a program executing on it. The specification is based on Data-Race-Free-1 (DRF1) memory consistency model [2], which has been extensively studied and is relatively easy for the programmer to reason with. Our work extends DRF1 to apply to highly integrated threads operating within the processor.

In this work we elaborate the programming model for practical application.

- For the source program, the correctness requirements are an integral part of the programming model.
- For the compiler, we define the requirements which ensure that the properties of the source code required by the programming model are preserved by the compiler optimizations, and prove the correctness of the requirements.
- For the microarchitecture, we define the requirements which ensure the processor obeys the requirements of the programming model and prove the correctness of the requirements.

We proceed to apply the correctness requirements to the implementation of the compiler and the processor. For the compiler, we show how to adjust the compiler’s inter-
mediate representation and the optimizations to fulfill the requirements. We devote especial attention to the process of register allocation, which is profoundly affected by register-shared multithreading. For the processor, we show how to implement the requirements under an in-order or an out-of-order pipeline.

One of the effects of register-shared multithreading is interference with the function calling mechanism. In this paper, we show the source of the incompatibility and describe the circumstances in which it is still beneficial to allow function calls during Inthreads-parallelized execution. For such calls, we develop a mechanism that allows issuing the calls without terminating the threads.

5.2 Inthreads Architecture

The Inthreads architecture is based on a fixed number of lightweight thread contexts which execute over a shared register file. The threads are responsible to use the registers in such a way as to avoid conflicts, dedicating a subset of registers for each thread’s private use. Furthermore, threads can communicate through the registers in the same way as conventional threads communicate through shared memory locations. To allow such communication to proceed safely, the architecture provides dedicated synchronization instructions. The instructions defined by the Inthreads Instruction Set Architecture (ISA) are summarized in Figure 5.1: three are for thread management, three for synchronization and two for function call and context switch support. For synchronization, the ISA defines a set of 1-bit condition registers which behave as binary semaphores. More information can be found in [30].

The Inthreads approach to thread management differs from that of the traditional SMT. SMT parallelization operates at runtime: the same code fragment can be assigned dynamically to several threads. In contrast, Inthreads multithreading explicitly designates individual code regions for the different threads. As a result, the Inthreads code generation process can directly manage the interactions between the threads.

The fixed nature of the Inthreads model allows for the thread scheduling and management to be performed entirely in hardware. Our current implementation supports 8 concurrent threads as a trade-off between the amount of expressible parallelism and the hardware complexity. Moreover, even if a program could be parallelized with more threads, the register pressure would become a bottleneck.

An important model for parallel code execution is Sequential Consistency (SC) [51]. An execution is sequentially consistent if it could result from interleaved atomic exe-
**inth.start**

```plaintext
id, addr
```

Starts a new thread with a given ID at a specified address.

**inth.halt**

Terminates the current thread. This instruction executes synchronously, guaranteeing completion of the preceding instructions.

**inth.kill tid**

Kills the specified thread. Unlike `inth.halt`, this instruction is asynchronous, i.e., there is no guarantee on the exact point where the thread will be terminated.

**inth.clr cond**

Clears the specified binary semaphore.

**inth.wait cond**

Suspends thread execution until the semaphore is set. The value of the semaphore is automatically cleared when the thread continues.

**inth.set cond**

Sets the value of the semaphore. If there are threads waiting on the semaphore, one of them is resumed and the value of the semaphore is cleared.

**inth.suspend**

Suspends the running threads and makes their context available for reading. Such functionality can be used by the operating system or by the compiler to suspend the parallel threads.

**inth.resume tid**

Restores the context of the running threads and continues executing in the given thread id.

---

Figure 5.1: Instructions defined by the Inthreads ISA

---

Excution of instructions of all the threads. SC is the most intuitive model for reasoning about parallel code, however, it is expensive to implement. Weak models allow for cheaper implementations by imposing constraints on permitted sequences of instructions.

The consistency used in the Inthreads model is based on *data-race-free* (DRF1) [2]. DRF1 distinguishes *strong* or *synchronization* instructions (further identified as paired *release* and *acquire* ones) from *regular* data accesses, and provides ordering guarantees between the strong and the regular instructions. DRF1 is defined on the basis of *happens-before*:

**Definition 5.1.** For each two operations \( o_1 \) and \( o_2 \) in an execution, such that \( o_1 \) is a
release, \( o_2 \) is an acquire and \( o_1 \) is paired with \( o_2 \), synchronization-order-1 (SO1) relation orders \( o_1 \) before \( o_2 \). Happens-before-1 (HB1) relation is the irreflexive transitive closure of the program order and SO1.

**Definition 5.2.** Two instructions are potentially concurrent if they are not ordered by the program order, and concurrent if they are not ordered by HB1.

**Definition 5.3.** Two instruction conflict if they access the same location and at least one of them is a write. A pair of concurrent conflicting instructions forms a data race. An execution is data-race-free if it contains no data races, i.e., any two conflicting accesses in it are ordered by happens-before-1. A program is data-race-free if all its sequentially consistent executions are data-race-free.

**Definition 5.4.** Hardware obeys data-race-free-1 if the result of any execution of a data-race-free program can be obtained for the same program on sequentially consistent hardware.

The Inthreads model maps the thread-related instructions to strong instructions: `inth.start`, `inth.halt`, `inth.set` and `inth.resume` instructions have release semantics, while `inth.wait` and `inth.suspend` have acquire semantics. These instructions are inherently paired through the execution: an `inth.set` is paired with an `inth.wait` it wakes up, and an `inth.start` is paired with the first instruction of the started thread. In contrast to the conventional DRF1, which considers the memory accesses only, the Inthreads model considers the register accesses as well.

The model prohibits data races in the programs altogether (unlike conventional models, in which the result of such programs is unspecified). While the implementation of DRF1 hardware is beyond the scope of this paper, we note that disallowing data races significantly simplifies the interactions between instructions of different threads, both for in-order [26] and for out-of-order [31] pipelines.

### 5.2.1 Inth-C

The Inthreads code generation process is conceptually divided into two stages, shown in Figure 5.2. The first stage, parallelization, expresses parallelism with explicit parallelization constructs. The second stage, code generation, translates the parallelized code into Inthreads-compliant machine language. The parallelization stage can be based on different levels of abstraction, including manual parallelization, semi-automatic parallelization through a parallelization framework such as OpenMP, or automatic one.

**Inth-C** is an intermediate language interfacing the two stages. It allows embedding regions of parallelized code anywhere in the program. A parallel region contains
code for each thread, syntactically enclosed within a single block and marked with a
\#pragma inthread directive. The thread starting, termination and synchronization is
performed by explicit commands in the code that correspond to the Inthreads ISA
instructions: \texttt{INTH\_START}, \texttt{INTH\_HALT}, \texttt{INTH\_KILL}, \texttt{INTH\_CLEAR}, \texttt{INTH\_WAIT} and
\texttt{INTH\_SET}. The commands are more than just placeholders for the corresponding
instructions: the compiler analyzes them to infer the behavior of parallel execution
and to enforce some constraints on the use of parallelization. Several examples of the
constraints are:

- The code blocks of all the spawned threads must be targets of \texttt{INTH\_START}
  commands.
- Every spawned thread must either stop with an \texttt{INTH\_HALT} or contain an
  infinite loop (in which case, another thread must terminate it with an \texttt{INTH\_KILL}).
- Jumps into or from the parallel code are prohibited.

Figure 5.3 shows an example of serial code and its Inth-C parallelized version.
Note that variable \(i\) is defined within the code block of each one of the threads and
therefore is thread-private, while \(\text{sum1}\) and \(\text{sum2}\) are visible to all the threads. A
synchronization between \texttt{INTH\_SET(1)} in \(T_1\) to \texttt{INTH\_WAIT(1)} in \(T_0\) is used to protect
the accesses to \(\text{sum2}\).

### 5.2.2 Shared Variables

In this paper we formulate the shared variables analysis in terms of scalar variables,
however, similar methods could be used to detect shared memory locations based on
memory aliasing analysis. Intuitively, a variable is \textit{shared} if it is used for communica-
tion, i.e., is assigned in one thread and later used in another. The definition is based
on definitions 5.2 and 5.6:

\textbf{Definition 5.5.} Variable \(v\) is shared if it is defined at some execution point \(e_1\) and
there exists another execution point \(e_2\) concurrent with \(e_1\), such that \(v\) is live at \(e_2\).
Note that our definition is slightly incompatible with the common one, in which a variable is shared if it is accessed from several concurrent threads. Some examples of the differences can be seen in Figure 5.4. In 5.4a, \( v \) starts to be live at the starting point of \( T_1 \). However, that point is not concurrent with the assignment \( v = 1 \) in \( T_0 \) because of the ordering imposed by \text{inth.start}. Intuitively, at the spawning of \( T_1 \), both threads inherit the value of \( v \), so \( v \) is not used for communication. In contrast, in 5.4b, \( v \) is written by \( T_1 \), and since the writing operation is concurrent with execution point \( e \), \( v \) is \textit{shared}. Intuitively, the value of \( v \) is communicated from \( T_1 \) to \( T_0 \), so it is shared — regardless of the fact that because of termination of \( T_1 \), the use of \( v \) does not belong to multithreaded code at all. Finally, in 5.4c, the variable \( v \) is used in both \( T_1 \) and \( T_2 \), but since every sequence of accesses to \( v \) starts with an assignment, no assignment statement is concurrent with a live range of \( v \). This is not surprising, as each thread obviously makes private use of the variable.

### 5.2.3 Compiler Correctness Requirements

To define the compiler correctness requirements, we use the following model of compiler operation. The compiler conceptually proceeds in two stages. During the first stage, roughly corresponding to the set of compiler optimization transformations, the
compiler can introduce new temporary variables and accesses to them. It can also introduce new statements, remove or reorder existing ones. During the second stage, corresponding to register allocation, the compiler maps the statements from the set of program variables \( V \) to the set of locations \( L \). At each statement, variable \( v_i \) can be mapped to some location \( l_i \), while no two different variables can be mapped to the same location. Throughout the program, several variables can be mapped to the same location or one variable can be mapped to several locations. While practical compilers perform some transformations after the second stage, these transformations are more restricted — for instance, no new variables are introduced.

Below we use the following definitions and notations. Program statement \( o \) in the original program is denoted by \( o^{SP} \), and in the compiled program by \( o^{CP} \). An execution point corresponding to \( o^{SP} \) is denoted by \( o^{SE} \), and an execution of \( o^{CP} \) is denoted by \( o^{CE} \). If a statement \( o \) was issued by thread \( T \), it is denoted by \( o \in T \).

The following requirements are sufficient in order to ensure correct compilation of context-sharing multithreaded code, assuming that the source code is data-race free.

**Requirement 5.2.1.** The compiler preserves the sequential execution semantics of each thread.

**Requirement 5.2.2.** The compiler avoids introducing accesses to temporary variables in such a way as to make them shared.

**Requirement 5.2.3.** The compiler preserves the order of synchronization instructions of each thread.

**Requirement 5.2.4.** The compiler preserves the order of accesses to the shared variables with respect to synchronization operations.

**Requirement 5.2.5.** The compiler avoids introducing accesses to variables that are not temporary, except for those directly appearing in the program source.

**Requirement 5.2.6.** The compiler avoids removing accesses to shared variables that...
are live at some synchronization statement.

**Requirement 5.2.7.** For two concurrent statements $e_1$ and $e_2$, if variable $v_1$ is defined at $e_1$ and variable $v_2$ is live at $e_2$, the compiler does not map $v_1$ and $v_2$ to the same location at $e_1$ and $e_2$.

**Requirement 5.2.8.** For every two concurrent statements $e_1$ and $e_2$, if variable $v$ is defined at $e_1$ and live at $e_2$, $v$ must be mapped to the same location at $e_1$ and $e_2$.

To prove the correctness, we first show that the requirements ensure that the generated code is data race-free, implying that the execution is sequentially consistent, and then claim that the execution results of the compiled code are possible under some execution of the source code.

**Theorem 5.1.** If the compiler obeys requirements 5.2.2, 5.2.3, 5.2.4, 5.2.5 and 5.2.7, then the code resulting from compilation of a data race free program is also data race free.

**Proof:** Assume that the execution of a compiled program contains two instructions $a_1^{CE} \in T_1$ and $a_2^{CE} \in T_2$ that form a data race accessing the same location $l$. Note that since the source program is data race free, there is no data race between the corresponding accesses in the code and the execution of the source program, i.e., between $a_1^{SP}$ and $a_2^{SP}$ and between $a_1^{SE}$ and $a_2^{SE}$.

Since $a_1^{CE}$ and $a_2^{CE}$ form a data race, one of them must be a write. Assume W.L.O.G. that $a_1^{CE}$ is a write access and $a_2^{CE}$ can be either a write or a read. The corresponding instructions in the compiled program $a_1^{CP}$ and $a_2^{CP}$ must also exist, $a_1^{CP}$ being a write. Consider the variables $v_1$ and $v_2$ that were used in the accesses in the source program from which $a_1^{CP}$ and $a_2^{CP}$ were generated. There are two cases:

**Case 1:** $v_1$ and $v_2$ are the same variable $v$. If $v$ is not a temporary one, requirement 5.2.5 implies that $a_1^{SP}$ and $a_2^{SP}$ must exist, with $a_1^{SP}$ being a write. Since the source program is data race free, every feasible path from $a_1^{SP}$ to $a_2^{SP}$ in the CFG must contain a sequence of synchronization instructions $a_1^{SP}(r_i, a_i) + a_2^{SP}$ such that $a_1^{SP} \rightarrow r_0$, $a_i \rightarrow r_{i+1}$, $a_n \rightarrow a_2^{SP}$ and $r_i$ is paired during execution with $a_i$. If $a_2^{SP}$ is a read, then $v$ is live at the statement immediately preceding $r_n$ and by definition 5.5, $v$ is shared. If $a_2^{SP}$ is a write, then, since $v$ is live at the statement immediately following $r_0$, $v$ is also shared. In both cases, because of requirements 5.2.3 and 5.2.4, a corresponding sequence consists in the compiled program between $a_1^{CP}$ and $a_2^{CP}$. Therefore, in any execution $a_1^{CE}$ and $a_2^{CE}$ would be ordered by HB1, contradicting the assumption that there is a data race between them.

If $v$ is a temporary variable, then, because of requirement 5.2.2, it must not be shared. However, the accesses to $v$ imply that it is defined at the $a_1^{SP}$ introduced by the compiler and live at $a_2^{SP}$. However, there can be no sequence of synchronization
instructions that would order \( a_{1}^{SP} \) and \( a_{2}^{SP} \) by HB1, and therefore, \( v \) was shared when introduced, violating requirement 5.2.2.

Case 2: \( v_{1} \neq v_{2} \), but the compiler has mapped them to a single location \( l \). According to requirement 5.2.7, the statements \( a_{1}^{SP} \) and \( a_{2}^{SP} \) did not conflict at the beginning of stage 2, i.e., the statements are not concurrent. Since the two statements are executed by two different threads, every path between them must contain a sequence of synchronization instructions ordering them by HB1. The processing of stage 2 does not change the order of accesses with respect to synchronization instructions, therefore, the same sequence of synchronization instructions must exist in the compiled program. We conclude that \( a_{1}^{CE} \) and \( a_{2}^{CE} \) are ordered by HB1 and thus cannot form a data race.

Q.E.D.

Since the compiled program is data-race-free, its execution on DRF1 hardware is sequentially consistent. It remains to show that the result of any computation performed by the compiled program is feasible under execution of the source program.

Consider the computation of some expression \( E \) in the program. If \( E \) does not involve input or output to shared variables, the sequence of operations involved does not interact with the computation performed by the other threads, and therefore, because of requirement 5.2.1, the result of the computation by the compiled program is the same as the one performed by the source program. If \( E \) has some inputs that are read from shared variables which are written by other threads, the writing operation and the reading one must be separated by a sequence of synchronization instructions. Because of requirement 5.2.6, the instruction writing to the variable in the producing thread and the instruction reading the variable for input to \( E \) are preserved during the compilation, and because of requirement 5.2.8, are mapped to the same location. Since the execution is sequentially consistent, the value written to the shared variable is yielded at the statement reading it. We conclude that the result of every expression computed by the compiled program is feasible under the original program, and therefore, the whole result of the execution of the compiled program is feasible under the source one.

Q.E.D.

5.2.4 Hardware Correctness Requirements

A synchronization sequence \( S_{a,b} \) between two instructions \( a, b \) is a sequence of the form

\[
a = a^0, o^0_1, \ldots, o^0_k, r^0, (a^1, o^1_1, \ldots, o^1_k, r^1)_{i \in 1 \ldots n-1}, a^n = b,
\]

where instruction \( a^i, \ldots r^i \) are instructions in the program order of the same thread and \( r_i \) is paired with \( a_{i+1} \). There can be multiple such sequences for a given pair of
instructions.

The processor implementation must observe the following requirements:

**Requirement 5.2.9.** The processor observes execution semantics of each thread in isolation.

**Requirement 5.2.10.** There is a sequentially consistent execution order of strong instructions that obeys the instruction pairing order and the program orders of the threads.

**Requirement 5.2.11.** For two accesses \( a \) and \( b \) such that \( a \) writes to some location \( v \), and \( b \) reads from the same location, if there exists at least one synchronization sequence \( S_{a,b} \) and no such synchronization sequence contains additional write accesses to \( v \), then the execution of \( b \) must yield the same variable written by \( a \).

**Theorem 5.2.** If the processor obeys requirements 5.2.9, 5.2.10 and 5.2.11, then execution of a program with no data races will be sequentially consistent.

**Proof:** We begin with \( S^0 \), the sequential order of the strong instructions and extend it to the required order \( S \) by adding to \( S^0 \) with the instructions that access the shared locations. For each thread, we consider the program order of its instructions. We split the program order into subsequences determined by the acquire and release instructions and process each subsequence independently. Let \( O_s^i = [o_1, \ldots, o_k] \) be the subsequence of instructions between the strong operation \( s_i \) and \( s_{i+1} \). We insert \([o_1, \ldots, o_k]\) to \( S \) immediately after \( s_i \).

The construction of \( S \) implies directly that the order of instructions in it is consistent with the program orders of the threads. To prove that the order is also sequentially consistent, we must show that each instruction \( a \) reading a variable \( v \) yields the value written by the latest instruction that wrote to \( v \) and preceded \( a \) in \( S \).

Consider two instructions \( a \) and \( b \) such that \( a \) is a write to a location \( v \) and \( b \) is a read from \( v \), and \( b \) yields the value written by \( a \). Assume by contradiction that either \( a \) does not precede \( b \) in \( S \) or there exists another instruction \( c \) between \( a \) and \( b \) in \( S \) that writes to \( v \). There are two cases:

1. \( a \) and \( b \) belong to the same thread \( T \). By requirement 5.2.9, \( a \) must precede \( b \) in the program order of \( T \), therefore, by construction \( a \) precedes \( b \) in \( S \). The only possible violation then is that there exists another instruction \( c \) in \( S \) between \( a \) and \( b \). Because of 5.2.9, \( c \) must belong to some other thread \( T' \neq T \). Since the program contains no data races, there must be a synchronization sequence ordering \( a \) and \( c \) and another ordering \( c \) and \( b \). If the synchronization sequence orders \( c \) before \( a \) (or after \( b \)), then by requirement 5.2.10 and by construction, the subsequence of instructions containing \( c \) would be inserted to \( S \) before the
subsequence containing $a$ (or after the subsequence containing $b$, respectively), violating the assumption. If the synchronization sequence orders $c$ after $a$ and before $b$, then the sequences form a longer synchronization sequence from $a$ to $b$ that contains a write access to $v$, violating requirement 5.2.11.

2. $a$ and $b$ belong to two different threads $T_1$ and $T_2$. Since the program contains no data races, there must be a synchronization sequence ordering $a$ and $b$. The sequence could not order $b$ before $a$, otherwise, requirement 5.2.11 would be violated. If another instruction $c$ exists in $S$ between $a$ and $b$, it will violate requirement 5.2.10 or 5.2.11, similarly to the previous case. Q.E.D.

### 5.3 Compilation of Explicitly Parallel Code

As a result of execution of threads in a shared context, many implicit assumptions taken by conventional compilers do not hold for Inthreads-parallelized code. For an example, consider the code in Figure 5.5. Variable $a$ is assigned in thread $T_1$ and is later read in the main thread. If a compiler processed the code of $T_1$ separately, it would not be aware of communication through $a$, and could erroneously identify the assignment to $a$ in $T_1$ as dead code, i.e., code that produces a value never used by subsequent instructions. It would then remove the assignment to $a$, generating incorrect code.

---

<table>
<thead>
<tr>
<th>(1) INTH_START(1,THREAD1);</th>
<th>(9) THREAD1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2) #pragma inthread</td>
<td>(10) #pragma inthread</td>
</tr>
<tr>
<td>(3) {</td>
<td>(11) {</td>
</tr>
<tr>
<td>(4)  b = ...;</td>
<td>(12) a = ...;</td>
</tr>
<tr>
<td>(5)  INTH_WAIT(1);</td>
<td>(13) INTH_SET(1);</td>
</tr>
<tr>
<td>(6)  a += b;</td>
<td>(14) INTH_HALT();</td>
</tr>
<tr>
<td>(7) }</td>
<td>(15})</td>
</tr>
<tr>
<td>T_0</td>
<td>T_1</td>
</tr>
</tbody>
</table>

Figure 5.5: Example of code that can be miscompiled by a threads-ignorant compiler.
5.3.1 Internal Representation

Most modern compilers use an internal code representation based on Control Flow Graphs (CFG) [61]. The nodes of a CFG are basic blocks of the program — sequences of instructions with linear control flow, while the edges represent possible control flow paths between the basic blocks.

A Concurrent Control Flow Graph (CCFG) [33] extends CFG with information on the parallel execution semantics. The CCFG used in the Inth-C compiler introduces two additional edge types. Parallel edges, which connect the outgoing edges of an INTH_START with the code of the started thread, are handled similarly to general control flow edges. Synchronization edges connect INTH_SET instructions with corresponding INTH_WAIT ones in parallel threads. The synchronization edges do not represent possible control flow paths, and are used only to analyze communication between threads.

When constructing the CCFG, the Inth-C compiler observes the following rules:

- Since a synchronization edge adds an outgoing edge after a INTH_SET instruction and an incoming edge before the corresponding INTH_WAIT, the basic blocks are split before each INTH_WAIT and after each INTH_SET to ensure that an INTH_WAIT is always the first instruction in its basic block, and an INTH_SET is the last one in its block.
- Since an outgoing edge is added for each INTH_START, the basic block is split after it to ensure the INTH_START is the last instruction of the block. An INTH_START, like a regular branch, has two outgoing edges, however, unlike a branch, both outgoing edges have 100% probability.
- An int.halt is the last instruction of a basic block without outgoing control flow edges.

Figure 5.6 shows the CCFG for the code in Figure 5.5. Two edges leave the INTH_START block to represent parallel threads, one representing the fallthrough path and the other — the thread being started. Thread 1 terminates with an INTH_HALT block with no outgoing edges. A synchronization edge connects INTH_SET(1) and INTH_WAIT(1) since they access the same condition register.

Note that since the thread boundaries are defined syntactically and all the threads but the main one are expected to terminate before the syntactically denoted portion of the main thread completes, the compiler can statically determine the boundaries of each instance of code parallelization. Moreover, thread execution cannot escape the function call frame and therefore, it is safe to analyze the multithreading of each function separately.
**Definition 5.6.** Variable $v$ is live [61] at execution point $e$ if there exists a path from $e$ to some use of $v$ that contains no other assignment to $v$. A variable $v$ is defined at execution point $e$ if the statement at $e$ assigns a value to it.

### 5.3.2 Shared Variable Identification

The algorithm that identifies the shared variables, IDENTIFYSHAREDVARS, is shown in Figure 5.7. The algorithm is based on the standard COMPUTELIVEINFO algorithm [61] that computes liveness information according to Definition 5.6. The algorithm updates the set of shared variables at every thread spawning point. However, the procedure COLLECTDEFSANDLIVE collects the data of all the threads spawned from the given point, and therefore, IDENTIFYSHAREDVARS considers all the pairs of threads. Function EDGELEAVESTHREAD determines if the given edge leaves the current parallelization instance.

If a variable is shared, then it is by definition live at one thread and defined at some point in another thread. It is easy to see that the computation of IDENTIFYSHARED-
Algorithm \textsc{IdentifySharedVars} (graph : CCFG)
ComputeLiveInfo(graph)
SharedVars = \emptyset
\textbf{for} block \in graph \textbf{do}
\textbf{if} block \equiv \textsc{INTH.START} \textbf{then}
def1, live1 = \textsc{CollectDefsAndLive}(block.succ[0]) # Figure 5.8
def2, live2 = \textsc{CollectDefsAndLive}(block.succ[1])
SharedVars = SharedVars \cup (def1 \cap (def2 \cup live2))
SharedVars = SharedVars \cup (def2 \cap (def1 \cup live1))
\textbf{end if}
\textbf{end for}
\textbf{return} SharedVars

Figure 5.7: Algorithm \textsc{IdentifySharedVars}

\textsc{VARS} will identify all the variables shared according to definition 5.5. However, since the algorithm operates on the program's CFG rather than on the sequence of the executed instructions, it is necessarily conservative. Further analysis algorithms consider interactions only along explicit pairs of \textit{release} and \textit{acquire} instructions, improving the precision.

5.3.3 Optimizations

Optimizations are code transformations performed by the compiler to increase the efficiency of the program. The optimizations can be generally categorized into \textit{data flow-sensitive} and \textit{data flow-insensitive} ones [61]. The data flow-insensitive optimizations, such as \textit{inlining} or \textit{jump optimization}, have little interference with parallel execution. In contrast, data flow-sensitive optimizations need significant adjustments.

Parallel Data Flow Analysis

The purpose of Data Flow Analysis (DFA) is to compute which pieces of information are available at given points of code. Typically, data flow information is generated or modified in the nodes of the CFG and propagated along its edges [61]. However, in presence of parallel execution, interactions can occur between instructions belonging
Algorithm CollectDefsAndLive (block : BasicBlock)
DefVars = LiveVars = ∅
for insn ∈ block do
    DefVars = DefVars ∪ insn.defined
    LiveVars = LiveVars ∪ insn.live
end for
for succ ∈ block.successors do
    if ¬ EdgeLEAVES_THREAD(block, succ) then
        D₁, L₁ = CollectDefsAndLive(succ)
        DefVars = DefVars ∪ D₁
        LiveVars = LiveVars ∪ L₁
    end if
end for
return DefVars, LiveVars

Figure 5.8: Algorithm CollectDefsAndLive (used by IdentifySharedVars in Figure 5.7)

to different threads, with no control flow connecting the interacting instructions. Therefore, data flow analysis needs to be adapted to correctly take parallel execution into account.

Recall that the source program is required to be data-race-free. Therefore, if two variable accesses conflict in the execution of the source program, there must be a sequence of paired synchronization instructions that orders the two accesses by happens-before-I relation. For any pair of synchronization instructions in the execution, there is a corresponding pair of synchronization instructions in the program’s CFG. Consequently, propagating the data flow information along synchronization pairs of the source program, i.e., along the synchronization edges, captures all possible data flow interactions between threads. Furthermore, we notice that interactions can only occur between shared variables, and therefore only data flow items concerning these must be propagated.

The propagation of dataflow through synchronization edges is conservative since some pairs of synchronization instructions will not interact under any execution. However, in Inthreads-parallelized programs, synchronization registers are often not reused

125

in different parts of a program and therefore, most synchronization edges are indeed feasible.

**Dead Code Elimination**

*Dead code elimination* (DCE) tries to remove code that performs useless computation. Formally, a statement is *dead* (as opposed to *essential*) if it produces a value never used in any of the following execution paths and has no side-effects. However, a value produced by one thread can be accessed by a different thread even when there is no sequential control path connecting the definition and the use. Hence, sequential version of DCE may misidentify some essential statements in parallel code as dead and remove them, violating requirement 5.2.6.

To adjust DCE for parallel code, it is sufficient to use the parallel data flow information. Since the set of computed live variables is a superset of the actual live variables according to definition 5.6, the compiler would not consider a variable used for communication as dead.

**Constant and Copy Propagation**

Given an assignment of the form $x = y$, *copy propagation* attempts to replace subsequent uses of $x$ with $y$. *Constant propagation* is a special case of copy propagation where $y$ is a constant. Both optimizations use data flow analysis to find the variable references that can be safely replaced. The effect of copy propagation is to repeat an access at a later point in the program order, potentially violating requirements 5.2.5 and 5.2.4.

Since the optimization moves accesses forward in the program order, it can only violate the *shared access* $\rightarrow$ *release* constraints of happens-before-1 but not *acquire* $\rightarrow$ *shared access* ones. Therefore, to preserve the correctness requirements, the compiler invalidates the values of the shared variables at each *release* operation.

**Common Subexpression Elimination**

*Common sub-expression elimination* (CSE) removes duplicated computation of the same expression by keeping the result of the first computation and replacing the subsequent recalculations with the stored result. Since CSE reuses earlier accesses at later execution points, its effect with respect to DRF1 is opposite to that of copy propagation: it introduces a private variable that keeps the result of an access to
a variable, essentially removing some accesses or moving them backward in the program order. As a result, CSE violates requirements 5.2.6 and 5.2.4. Note that since it moves instructions in the opposite direction, it can only violate the acquire $\rightarrow$ shared access part of requirement 5.2.4. To adapt CSE for parallel execution, the compiler invalidates the data on expressions containing a shared variable at each acquire operation.

### 5.4 Register Allocation

Most modern compilers use virtual registers to represent user variables and temporary values in the intermediate code. The number of virtual registers is unlimited, simplifying the implementation of machine-independent optimizations. After those optimizations, compiler performs register allocation—mapping the virtual registers to the limited set of architectural registers available on the target processor.

The dominant approach to register allocation, graph coloring, generates register assignment for the whole CFG. The idea of the algorithm is to represent the problem by an interference graph, in which the nodes represent virtual registers and the edges connect any two nodes that conflict, i.e., must be allocated to different registers. The register allocation is then translated to $K$-coloring of the graph, where $K$ is the number of available architectural registers. Alternative approaches to register allocation, such as that taken in the GCC compiler [93], are known but are of less importance.

The graph coloring problem in general is known to be NP-complete. Chaitin’s register allocator [14] uses a heuristic approximate algorithm. The algorithm performs a series of simplification passes. Each pass proceeds in stages shown in Figure 5.9.

The stages are:

**Build** Construct the interference graph from the CFG.

**Coalesce** Non-interfering variables are coalesced to increase the chance that they will use the same color.

![Heuristic Graph Coloring Algorithm Flow](image)
Simplify Unconstrained nodes, those that have less than $K$ neighbors in the graph, are removed—a valid coloring for those nodes is guaranteed. If all the nodes are constrained, the algorithm removes one node heuristically, and marks that node as a spill candidate.

Select Colors are assigned to nodes in the reverse order of their removal. If all the spill candidate nodes are colored successfully, the allocation is found and the algorithm completes. Otherwise, the nodes for which coloring was not found are marked for spilling and the algorithm continues to the next stage.

Spill Nodes that could not be colored are spilled, i.e., are assigned to a memory location rather than to a register. This simplifies the interference graph, increasing the chance that the next pass will find a coloring.

There are two variants of the coloring-based register allocation. A basic variant uses each variable as a node in the interference graph. A more refined one, used in most modern implementations, uses register ranges, also called webs [61]. A register range represents an independent subset of accesses a virtual register and can be seen as an independent variable. Figure 5.10 shows register ranges in an example program. A single virtual register $a$ is split into two register ranges ($a_1$ and $a_2$). Each register range is then treated independently during register allocation run and can be allocated with a different machine register.
5.4.1 Register Allocation for Concurrently-Executing Code

Conventional computational models assume that each thread has a private register file and call stack, and therefore, the register allocation needs not take the interactions between threads into account. In contrast, this assumption does not hold for the Inthreads model. Since the register allocation performs mapping from the set of variables to the set of locations, the correctness requirements that can be affected by it are 5.2.7 and 5.2.8.

If the register allocation algorithm operates at a resolution of virtual registers, requirement 5.2.8 is trivially satisfied, since each variable can be mapped only to a single register. The correctness of register range-based allocation is less straightforward. We need to show that if two accesses to a variable can communicate, then they will necessarily be assigned to the same register range. Therefore, if register allocation considers the two accesses in the same pass of register allocation, they will be allocated to the same register.

To show that potentially communicating variable accesses are assigned to the same register range, we recall that the source program is required to be data-race-free. Any two accesses \( w, r \) issued by different threads but address the same variable \( v \) are conflicting unless ordered by a sequence of release-acquire synchronization instructions. Since the construction of the concurrent control flow graph, described in 5.3.1, includes an edge between any two pairable synchronization instructions, a path between \( w \) and \( r \) exists in the CCFG. Consequently, it is sufficient to use the parallel data flow described in Section 5.3.3 when computing the register ranges.

Explicit Register File Partitioning

One approach to satisfying correctness requirements is to pre-partition the register file, allocating a private subset of registers for each thread, and leaving a subset of registers for variables used for communication. Given the partitioning, the register allocation can be executed for the code of each thread independently to allocate the thread-private variables, and once more for the shared variables only.

Pre-partitioning based allocation trivially ensures the correctness of generated code. Requirement 5.2.7 follows immediately from using disjoint subsets of the register file for each thread. Requirement 5.2.8 follows since we have shown that all variables used for communication are identified as shared and will therefore be allocated during the same register allocation pass, the one that manages the shared variables.

While the correctness of the algorithm is straightforward, its implementation is
not. The reason is that optimal partitioning needs information on register pressure in the threads. However, since this information becomes available only after coloring, a circular dependency is induced between the partitioning and the coloring.

One way to avoid the circularity is to let the user control the partitioning. However, this approach is undesirable since it is too complex and time consuming: the user needs to evaluate the register pressure manually and to be aware of the details of the architecture of the target machine. Alternatively, the compiler could proceed iteratively, adjusting the partitioning in order to optimize the final register allocation. This approach, while automatic, would be computationally expensive.

**Implicit Register File Partitioning**

In order to avoid explicit partitioning of the register file, we use the parallel CFG to construct the register ranges as described in Section 5.4.1. We then construct a parallel interference graph using a modified version of the conventional algorithm. Finally, we use a conventional coloring algorithm, obtaining register allocation for all the threads at the same time.

The algorithm, shown in Figure 5.11, differs from the conventional one in two aspects. First, it uses the CCFG for liveness information. Second, it considers all pairs of potentially concurrent accesses as interfering.

Requirement 5.2.8 follows immediately from executing the coloring in one pass. To show that requirement 5.2.7 is satisfied, recall that the parallel interference graph contains an edge between any two potentially concurrent accesses performed by different threads. Since the coloring assigns different registers to variables connected by an edge, private variables used by different threads will use different registers.

Considering all pairs of potentially concurrent instructions as interfering is conservative, and the accuracy could be improved by a more precise implementation of \texttt{GetConcurrentStatements}. Nevertheless, our approach produces near-optimal results for most of the parallelization cases. The reason is that Inthreads parallelization tends to minimize the amount of synchronized code. Therefore, the conservative evaluation is accurate for the majority of the code of the threads.

**5.4.2 Spill Code Generation**

When the number of machine registers is insufficient to accommodate all the virtual registers, the compiler performs spilling by assigning some of the variables to stack locations. For such variables, the compiler generates spill code, storing the value to
Algorithm \textsc{CollectRegConflicts} (ccfg):
\[\text{Conflicts} = \text{CollectSequentialConflicts}(ccfg)\]
for \(E_1, E_2 \in \text{GetConcurrentStatements}()\) do
  for \(\text{regrange}_1 \in \text{Defined}(E_1)\) do
    for \(\text{regrange}_2 \in (\text{Live}(E_2) \cup \text{Defined}(E_2))\) do
      if \(\text{regrange}_1 \neq \text{regrange}_2\) then
        \(\text{Conflicts} \cup = (\text{regrange}_1, \text{regrange}_2)\)
      end if
    end for
  end for
for \(\text{regrange}_2 \in \text{Defined}(E_2)\) do
  for \(\text{regrange}_1 \in \text{Live}(E_1)\) do
    if \(\text{regrange}_1 \neq \text{regrange}_2\) then
      \(\text{Conflicts} \cup = (\text{regrange}_1, \text{regrange}_2)\)
    end if
  end for
end for
\(\text{return} \ \text{Conflicts}\)

Figure 5.11: Algorithm to collect intra-thread register range conflicts

the stack to free the register and loading it when necessary.

In the simplest method, \textit{spill-everywhere} [14], a spill candidate is spilled immediately when encountered. For a spilled register range, \textit{stores} are inserted after each definition and \textit{loads} are placed before each use. Later proposals introduced improvements to this basic behavior, such as \textit{optimistic coloring} [9] which delays spilling until the \textit{Select} phase, and \textit{interference region spilling} [8] that spills only part of the accesses.

For parallel code, spilling shared variables requires additional considerations to preserve correctness. Since it introduces several additional locations for the same variable and the corresponding access instructions, it can potentially violate require-
ments 5.2.5 and 5.2.4, 5.2.2 and 5.2.8.

Spill Everywhere

*Spill everywhere* processes each basic block independently. For each variable selected for spilling, the algorithm inserts a *store* after each definition and a *load* before each use. A new temporary is used for every spilled access.

Essentially, the method introduces multiple locations for every spilled variable: a *permanent* one, the stack slot reserved for the variable, and several *temporary* ones in registers. Requirement 5.2.2 is preserved since a different virtual register is used for each spilled access. Since the sequence of accesses to the permanent location is the same as the sequence of accesses to the original variable, spill-everywhere preserves the rest of the correctness requirements as well.

One optimization over the basic spill everywhere avoids unnecessary loads between close references to a spilled variable [61]. Subsequent references to a variable are *close* if no other variable ceases to be *live* between them and thus freeing a register through spilling would not be of any use. To see that this approach is still valid, recall that basic blocks of the CCFG are split on synchronization instructions. Since spill-everywhere is performed at the basic block granularity, the optimization still preserves requirements 5.2.4 and 5.2.6.

Interference Region Spilling

One drawback of the spill everywhere method is its coarse granularity: if a variable is selected for spilling, all the accesses to it must be performed through memory. However, spilling might be unnecessary in some regions of the graph where the register pressure is low. *Interference Region Spilling* addresses this problem by spilling only a subset of accesses to a variable.

The algorithm operates on pairs of variables. If a variable $v_1$ needs to be spilled, it selects another variable $v_2$ conflicting with $v_1$. Then, in regions where only one of the variables is live, it can be placed in a register with no need for spilling. In regions where both are live, the standard spill-everywhere method is used. At a boundary between such regions, where $v_1$ continues to be live but $v_2$ does not, $v_1$ must be *reloaded* from its permanent location to a register. Similarly to spill-everywhere, interference region spilling is safe for private variables.

Since interference region spilling is applied only to a subset of accesses, it essentially increases the live ranges of the spilled variables, which can cause a reload operation to switch order with a synchronization instruction, violating requirement 5.2.4.
Furthermore, several locations assigned to a variable might be shared between threads. In such cases, reloads from one location to another are effectively additional write accesses, violating requirements 5.2.2 and 5.2.5.

Examples of problems with Interference Region Spilling can be seen in Figure 5.4.2. Mitigating the effects of interference region spilling requires shrinking of variable live ranges to avoid crossing synchronization instructions and using a different private variable for each reload statement. These measures were not used since they effectively constrain the spilling to operate as spill-everywhere.

In our implementation, we use a combination of the two methods: spill-everywhere is applied to the shared variables, while the rest of variables are handled through interference region spilling.

### 5.4.3 Coalescing

Coalescing is a step in register allocation that merges non-interfering register ranges [5]. The merging introduces a node with a potentially high number of interference edges, however, it still might be profitable when the ranges originate at the same variable: in that case, mov instructions that copy the Valle from one register range to another can be eliminated.

Coalescing is always safe for parallel code. To see why, consider two register ranges \( a \) and \( b \) that were coalesced, renaming \( b \) to \( a \), and caused a data race. Since the only change in the CFG is that of replacing accesses to \( b \) with accesses to \( a \), the data race has to be between two accesses \( e_1 \) and \( e_2 \) to \( a \) performed by two different threads,
such that before the coalescing, one of \( e_1, e_2 \) accessed \( b \) and at least one of \( e_1 \) and \( e_2 \) is a write. Since \( e_1 \) and \( e_2 \) are issued by different threads, the interference graph would contain an edge between \( e_1 \) and \( e_2 \), and therefore, \( a \) and \( b \) would not be candidates for coalescing.

### 5.4.4 Register Allocation Success Guarantee

Coloring-based register allocation is guaranteed to succeed for single-threaded execution, regardless of the complexity of the CFG. The reason is that the algorithm eventually spills all the variables, resulting in a degenerate interference graph that can be colored with just three colors (assuming that each instruction has at most two inputs and one output).

This reasoning does not hold for a parallel interference graph. Even when all the register ranges in the program are spilled, the graph will contain interference edges between variables of different threads. If the graph is sufficiently complex, all the nodes will be constrained, and the heuristic coloring algorithm will fail.

Still, a conservative parallel interference graph is always colorable. To see why, note that similarly to how the serial interference graph is colorable with three colors, a parallel interference graph of \( k \) threads is colorable with \( 3k \) colors after maximal spilling. Since two variables accessed by different threads are considered interfering, a greedy algorithm would never use the same color for them. Since under maximal spilling, the different variables of the same thread do not conflict with each other, the algorithm would use the same three colors for all the nodes belonging to each thread, and therefore, would succeed to color the graph in \( 3k \) colors.

The register allocation algorithm must be adjusted to take advantage of the guaranteed coloring. The problem is that selection of spill candidates contains a short register range heuristic. Short register ranges are such ranges during which no other register dies, and therefore, the register freed by spilling them could not be reused. The conventional algorithm never considers spilling such ranges since it is impossible for all the ranges to be short and constrained. However, this is possible with a parallel interference graph. To this end, if all the register ranges are short and constrained, our algorithm selects one random range for spilling and proceeds.

The greedy algorithm may fail coloring a less conservative interference graph. For an example, consider the graph in Figure 5.13. Nodes \( v_i \) represent variables used by one thread and nodes \( u_i \) — by another one. The graph has no edges connecting \( v_i \) to \( u_i \); this could result, for instance, from mutual exclusion synchronization between \( v_i \) and \( u_i \). It is easy to see that the graph is bipartite and therefore colorable. However,
Figure 5.13: Interference graph that causes greedy coloring to fail

if the graph is colored in the order \( u_0, v_0, u_1, v_1, \ldots \), the number of colors used by the greedy algorithm is unbounded: \( u_0 \) and \( v_0 \), which do not conflict, would get color 0, \( u_1 \) and \( v_1 \) would have to use color 1, and so on.

We can ensure coloring for any parallel interference graph by controlling the order in which colors are assigned. To this end, if coloring fails, we perform coloring for all the register ranges of a thread before starting to color the edges of another one. This procedure is guaranteed to eventually succeed since under maximal spilling, no communication between threads is done through registers and each thread’s part of the graph is colorable in three colors.

5.5 Microarchitecture Implementation

In this section we consider the requirements stated in 5.2.4 for two major pipeline organizations: inorder and out-of-order. In order to focus on the discussion, we use models of the processor implementation. Certain implementation details, such as the exact specification of the control lines, have been omitted. Furthermore, we have omitted the discussion of speculative instruction execution; this was described in detail in [31].

5.5.1 Inorder Pipeline

An inorder pipeline can work on several instructions in parallel, but cannot change the order between them. Each instruction is first fetched and decoded. It can be issued when all its resources (input values and processor’s functional units) become available and all the preceding instructions are issued. An issued instruction is executed and disappears after applying its side effect to the processor’s state. Figure 5.14 shows the extensions that support Inthreads execution in an inorder pipeline.
Figure 5.14: Pipeline model for an inorder processor

- The *Fetch* stage is extended to fetch instructions for several threads during each cycle, mix them and send down the pipeline.

- No changes are necessary in the instruction decoding.

- We add a special *Wait* stage is added to delay `inth.wait` instructions prior to execution. Note that `inth.set` instructions are executed at a further stage of the pipeline.

- The *Issue* stage is extended to independently issue instructions of different threads, allowing instructions of one thread to proceed while instructions of another are waiting on some resource.

- Only minor changes are necessary in the instruction execution, none are relevant for the correctness discussion.

Requirement 5.2.9 follows since the subset of the pipeline dealing with instructions belonging to each thread involves exactly the same processing steps as the original pipeline.

Since the pipeline processes instructions in order, the order in which the strong instructions are issued is consistent with the threads’ program orders as well. Note that the effects of *release* instructions are produced at the *Execute* stage, and the effects of *acquire* instructions are produced at earlier pipeline stages, *Fetch* for `inth.start` and *Wait* for `inth.wait`. Therefore, strong instructions issued at the same cycle cannot affect each other and can be ordered arbitrarily. We conclude that the above order of strong instructions will be also consistent with their pairing semantics and thus, the sequence satisfies requirement 5.2.10.

Requirement 5.2.11 follows from the fact that the program is free from data races. If two threads $T_1$, $T_2$ contain two conflicting instructions $a$ and $b$, respectively, accessing the same location $v$, then there must be a sequence of synchronization instructions
ordering $a$ and $b$. Assume W.L.O.G. that the sequence orders $a$ before $b$. Since the processor preserves the program order of each thread, the first release instruction following $a$ in its program order will be executed after $a$. Since acquire instructions are processed at an earlier pipeline stage than release ones, $b$ will arrive to Issue by the time $a$ has begun executing. If no other instruction wrote to $v$ between $a$ and $b$, the processor’s sequential execution semantics will ensure that $b$ will yield the value supplied by $a$. Q.E.D.

5.5.2 Out-of-Order Pipeline

An out-of-order (OOO) pipeline can execute instructions as soon as their resources become available, without necessarily waiting for the preceding instructions to issue. To this end, the pipeline introduces a Rename stage, in which false dependencies are eliminated by assigning a new location for each generated value. In addition, the queue in the Issue stage can issue ready instructions from any position. Figure 5.15 shows the extensions that support Inthreads execution in an OOO pipeline.

- The Fetch unit is extended to fetch instructions from all the threads simultaneously. Each cycle, it can provide an arbitrary mix of instructions of different threads.

- Decode, Rename, Issue, Execution and Commit do not require conceptual changes to support Inthreads.

- A Wait stage is inserted between Decode and Rename that delays instructions
following a `inth.wait` that is waiting on some condition. The delayed instructions are stored in an individual `Wait Buffer` for each thread.

- All the thread-related instructions are sent to the `Thread Control Unit` (TCU). The TCU executes all the received instructions except for `inth.wait`s. The rest of instructions are executed on the regular pipeline.

Note that a single issue queue for all the threads is necessary in an OOO pipeline. The difference with inorder pipeline is that the OOO one can switch order between an instruction and a `inth.set` following it in the program order, possibly violating the ordering of instructions connected by a synchronization chain.

Requirement 5.2.9 follows since regular computation instructions belonging to a single thread proceed through exactly the same computation stages as those in the original pipeline. Requirement 5.2.10 is a direct result of the organization of the TCU, which processes instructions in their arrival order.

To see that requirement 5.2.11 holds, note that the pipeline establishes register dependencies at the `Rename` stage. While the processor can change the order of instruction execution after `Rename`, it obeys the dependencies discovered there. Since the programs are required to be data race-free, any two potentially conflicting instructions $a$, $b$ must be ordered by a synchronization sequence. Since all the `release` instructions of the sequence are executed at the TCU, and the `acquire` instructions are executed at earlier stages (`Wait` for `inth.wait`, `Fetch` for the started threads), $a$ will arrive to `Rename` before $b$, and therefore, the processor would preserve the dependencies between them.

A similar reasoning applies to the ordering of accesses to memory-based locations.

5.6 Function Call Support

Single-threaded function calling conventions are based on an implicit assumption that the calling function is inactive while the called function is being executed. Under this assumption, it is sufficient to save the caller’s context on the stack before performing the call and to restore the context after the call completes. Unfortunately, the assumption is incorrect for threads executing over shared registers: if one thread calls a function and the rest of the threads, which operate in the call frame of the caller function, remain active, the register accesses of the called function can interfere with the accesses of the active threads.

An example of the interference is presented in Figure 5.16. Threads $T_0$ and $T_1$ are executing in the call frame of function $f$. There are no conflicts in the use of registers.
Figure 5.16: Register interference resulting from a function call

as $T_0$ uses registers $r0, r1, r2$ and $T_1$ uses $r3, r4, r5$. Consider the call to $g$ by $T_0$. If $T_1$ was not active, it would be sufficient to save and restore $r2$ for the duration of the call. However, since $T_1$ continues to operate, its accesses to $r3, r4$ will conflict to those performed by $g$ in thread $T_1$.

We have used two approaches to handle function calls under Intthreads parallelization. The first approach is to eliminate a function call altogether by inlining. Unfortunately, inlining is not universally applicable: for example, it cannot handle system calls, code from libraries and recursive functions.

Another approach is to suspend threaded execution for the duration of a function call, allowing the call to proceed according to the standard calling convention. This method introduces serialization and will be applicable only to parallelized code that performs function calls only rarely. However, such situations arise frequently with Intthreads parallelization, for example:

**Resource management** If a program keeps a buffer or a pool of some resource received from the OS, such as memory or file data, most accesses would be satisfied from the pool, performing OS calls only when the pool is exhausted. In this scheme, the majority of the pool accesses could be made from parallelized
code, and switches to single-threaded mode would be rare.

**Error detection** Some programs contain a considerable amount of error handling code, which is deeply integrated with the computation but almost never executed. Such error handling usually involves function calls and would by itself prevent parallelization. In cases where it is the only factor that prevents the use of Inthreads, suspending would enable parallelization at very little overhead.

**Partial inlining** In some cases, a function frequently proceeds through a simple, frequently executed code path and only reaches complex code for rarely occurring inputs. Such functions would be extremely expensive or even infeasible to inline entirely, while inlining of just the frequent part would be beneficial. In such cases, we could split the function into the frequent case code which calls a separate function for the rarely executed case, using inlining for the former and suspending for the later. This would allow us to apply Inthreads parallelization to the majority of dynamically executing code.

Examples are trigonometric functions from the *math* library.

### 5.6.1 Suspend/Restore Implementation

There are two approaches to switching between multithreaded and single-threaded execution. In the simpler *automatic* switching, the processor detects all function calls during multithreaded code and switches to single-threaded mode for the duration of the call. This approach, while being reliable and simple for the software, has several drawbacks. One problem is a significant increase in hardware complexity. Another issue is that the software would have no control over the granularity of switching. The approach taken in our implementation is for the software to explicitly switch the execution mode between single-threaded and multithreaded.

The second approach, described in the rest of this section, is to require the software to explicitly request the processor to suspend multithreaded execution.

**Hardware Support**

The actions involved in switching execution to single-threaded mode are shown in Figure 5.17. Under normal operation, the processor enters multithreaded mode by issuing one or more *inth.start* instructions, and leaves the mode when all the started threads terminate due to *inth.halt* or *inth.kill* instructions. These transitions are shown
Figure 5.17: Processor state machine for switching between multithreaded and single-threaded modes

with dashed arrows. Alternatively, temporary switch from multithreaded to single-threaded execution is initiated by `inth.suspend` and restored by `inh.resume` instructions.

As with the design of the Inthreads architecture, we require the software to take care of execution consistency. To this end, the `inh.suspend` and `inh.resume` instructions perform just the deactivation and reactivation of the threads, while the saving and restoring of the context data is done in the code.

Note is that the thread that issued the `inh.suspend` will not be suspended, and therefore, its PC is not static and cannot be saved and restored. Furthermore, since the called function should be unable to detect the fact that a `inh.suspend` was performed, the thread executing after a `inh.suspend` should have TID=0. To this end, we switch the TID of the running thread to 0 during execution of a `inh.suspend`. The `inh.resume` instruction should switch back the execution from TID=0 to the thread that performed the `inh.suspend`. To avoid storing the TID and retrieving it, we use the fact that the association of code with the threads is determined at compile time: the `inh.resume` instruction includes the TID of the thread to be restored as a parameter.

Several additional issues need to be considered in the execution of `inh.suspend` and `inh.resume`:

- A `inh.suspend` might reside on a misspeculated path. To avoid restoring threads in case of a squashed `inh.suspend`, we do not execute it speculatively. By similar
reasoning, a `inth.resume` should not be executed speculatively either.

- **Interactions with synchronization instructions might cause a deadlock.** If a `inth.suspend` follows a `inth.wait` in the program order of its thread, and the `inth.wait` needs a `inth.set` to be executed by some other thread, executing the `inth.suspend` might prevent the `inth.set` from being fetched altogether, thus causing a deadlock.

- **Instructions of other threads left in the pipeline may cause inconsistencies.** If the pipeline contains `inth.set` or `inth.clr` instructions when a `inth.suspend` is executed, the saving of condition registers will race with updating of them by `inh.set`. Therefore, the saved values of condition registers might be inconsistent with the execution. The same problem occurs with branches pending in the pipeline racing with saving of the PC registers of the threads.

To prevent the problems above, we delay an `inth.suspend` in the *Wait* stage until no other instructions remain the pipeline. The only exceptions are `inth.wait` instructions of the threads, which might be stuck in the *Wait* stage and would be unable to proceed until the thread that executed the `inth.suspend` completes the function call. To this end, we squash all the unexecuted `inh.wait` instructions, causing them to re-execute after the function call. When the `inth.suspend` is ready to execute, the processor switches itself to single-threaded mode. In that mode, the values of the PCs and the condition registers are available, so the software retrieves them and stores on the stack.

The delaying of `inth.resume` instructions is also performed by the *Wait* stage.

To establish correctness of the suspend/resume mechanism with respect to the requirements, we note that `inth.suspend` instruction is delayed until the processor contains no other active instructions. Therefore, when it is executed, all the preceding `inh.wait` instructions in the same thread have been completed and all the synchronization dependencies have been obeyed. Similarly, since a `inth.resume` is delayed in the *Wait* stage, before the instructions are executed out-of-order, any ordering dependency between the `inth.resume` and any following `inth.sets` are kept.

**Compiler Support**

The compiler support for function calls in Inthreads-parallelized code must intervene at several stages of code generation, as shown in Figure 5.18. First, the compiler identifies the instances of function calls in parallelized code and denotes the relevant places with `suspend` and `resume` markers. In the simplest case, the markers are
placed automatically around every call. However, in order to reduce the overhead of switching between single-threaded and multithreaded execution, the compiler joins sequential blocks of function calls to form continuous protected sections. Furthermore, the programmer can enlarge the protected sections by inserting the markers manually.

The compiler processes the markers at a late stage in the code generation, basically after register allocation. Initially, it prepares each call frame that contains a suspend/resume sequence by allocating a dedicated region for context data on the stack. The region will be used to store the thread PCs and the condition registers only; the additional saved registers are written to the stack in the same way as the regular ones.

A **suspend** marker is translated into the following sequence:

1. An `inth.suspend` instruction.

2. **Saving of the extra registers.** The compiler must take into account the registers used by the concurrently executing threads and save them in addition to the registers required by the calling convention. As an optimization, the compiler performs the additional register saving once per protected block rather than for every function call.

3. **Writing the function parameters to registers.** Since the registers required for parameter passing might be used by concurrently executing threads, the parameter passing must be done only after the threads are suspended and the registers have been saved.

A **resume** marker is translated into the following sequence:
Figure 5.19: Speedup of parallelization as a function of the register file size

1. **Copying of the return value to assigned location.** Similarly to the parameter passing in `suspend`, the return value location specified by the calling convention might not be available because of interference with other threads. In such case, the return value of the last function call in the protected region must be copied to its permanent location.

2. **Register loading** — the counterpart of the register saving in the `suspend` section.

3. An `inth.resume` instruction.

### 5.7 Evaluation

We have implemented the Inth-C compiler and the optimizations described in this paper on the basis of the GCC compiler [93]. The implementation involved both changing the front-end to recognize the Inth-C parallelization constructs and adaptation of the internal representation and code optimizations as described above.

To evaluate the Intthreads parallelization performance we constructed a simulator based on SimpleScalar toolset [11]. We have used the PISA instruction set, which is an extension of the MIPS ISA that allowed us to experiment with a large number of architectural registers. The evaluation used a set of benchmarks, including three benchmarks from Spec2000 suite [37]: Art, Mcf and Twolf, and most of the benchmarks from Mediabench suite [52].

Figure 5.19 shows the effect of the register file size on speedup achieved by the
Figure 5.20: Serial code slowdown as a function of the register file size

benchmarks compared to that of the original, non-parallelized, version. The result depends on the register pressure in the original code. For benchmarks with relatively complex code, such as Mesa, Epic and Art, performance improves significantly with the increase in the number of general purpose registers, and the rest of the benchmarks are almost unaffected. Indeed, when we compare the effect of register file size on the execution speed of the original (Figure 5.20a) and the parallelized (Figure 5.20b) code, we see that the only significantly affected serial program is Mesa, which is slowed down by 30% on 12 registers. The parallelized programs are more sensitive to the register file size, with Mesa slowing down by more than 50%.

Note that Mesa is the only benchmark that is slowed down by parallelization given a 32-GPR register file size. We conclude that Inthreads parallelization with automatic code generation is feasible for currently existing RISC architectures, and can benefit from increasing the number of architectural registers to 64.

5.8 Related Work

The syntax of Inth-C reminds that of Parallel Fortran, especially the Parallel Sections Statement. A significant amount of research has been dedicated to compilation of parallel code. Dirk and Srinivasan have treated data flow analysis of parallel programs [33]. The analysis was performed on the basis of Parallel Sections construct of PCF Fortran extensions. The major difference between Inth-C and PCF Fortran is that the Inthreads design is a combination of the architecture and the programming
language.

In addition, Inth-C is related to OpenMP [17]. The major difference between the two languages is that OpenMP operates in a dynamic manner, assigning work to threads and even deciding on the number of threads during runtime. Many of the OpenMP directives can actually be compiled into Inthreads ISA, therefore, we see Inthreads as complementing OpenMP rather than as an alternative to it.

A lot of research was invested into compilation of explicitly parallel code, from parallel program analysis [72, 48, 49, 78, 80] to explicit support for optimizations [47, 53]. The analysis performed by Inth-C is in many cases significantly simpler. One factor that contributes to that is the significantly lower parallelization granularity and, as a result, lower complexity of the parallelized code. In addition, the DRF1 contract and the data-race-free requirement simplify the analysis by restricting the compiler to dealing only with well-behaved programs.

The issue of compilation for register-sharing architectures has received some attention. Redstone [67] proposes sharing the register file between concurrently executing threads, however, the registers are divided statically and are not used for communication. [87] describes a more fine-grain register allocation between threads, however, the task is much simpler when registers are never shared between threads.

## 5.9 Conclusions

The Inthreads architecture achieves its efficiency from synergistic collaboration between the hardware and the software components. For the synergy to happen, we need a precise programming model defining the division of responsibilities between the components that utilizes the unique strengths of each one. In the case of Inthreads, we place the responsibility of ensuring consistency on the software, while the hardware support consists of ordering guarantees for several specially provided instructions.

This paper includes both the theoretical and practical sides of the Inthreads programming model. On the one hand, we provide a formal specification of the correctness requirements of the source program, the compiler and the processor. On the other hand, we show how to implement the requirements in the concrete frameworks of compiler optimizations and processor microarchitecture. We have shown that the model reduces complexity of both the software and the hardware by relying on the assumptions on the other side’s behavior.
Chapter 6

Summary

The basic idea underlying Inthreads is simple: to bring the ideas and the techniques from the domain of shared memory programming to the level of processor architecture. To this end, the architecture must trade off some of the flexibility in favor of simplified hardware implementation. In particular, it shares the thread execution contexts to the maximal extent possible and assigns the responsibility of avoiding conflicts entirely to the software.

The architecture is based on a programming model which requires active support on the software side. The processor provides a set of synchronization primitives; the responsibility of the software is to use these primitives to ensure the absence access conflicts. By requiring a collaboration between the program and the hardware, the programming model simplifies the implementation of both. As shown below, the correctness of compiler optimizations is ensured only for programs complying to the programming model. Furthermore, the requirements of the programming model allow us to reduce the size and complexity of hardware structures and contribute to the processor’s energy efficiency.

This work has examined the implementation and application aspects of the Inthreads architecture, including the programming model, the compilation process and the microarchitecture. To aid programming, we have developed Inth-C programming language 5.2.1, which is an extension of the C language with explicit parallelization. Inth-C is able to express many of the parallelization patterns, and includes enough parallelism information to be compiled directly into Inthreads machine code. Our Inth-C compiler is able to automatically analyze the parallel code to distinguish the shared variables from the private ones, using that information to guide optimizations and register allocation. From the microarchitecture side, we have shown how to
efficiently implement Inthreads under both inorder and out-of-order pipelines.

Chapter 2 presents an initial evaluation of the Inthreads architecture. The evaluation was performed prior to the implementation of the Inth-C compiler and therefore used only microbenchmarks and small excerpts of code of real benchmarks. The paper discusses and evaluates different parallelization patterns: loop partitioning (Figure 2.4, 2.5), virtual loop pipelining (Figure 2.7), speculative job starting and load-balanced worker pool-based job assignment. The microbenchmarks have achieved near-linear speedup up to the processor resource limit (Figures 2.6, 2.8, 2.10), and excerpts of SPEC benchmarks have achieved 10%-90% speedup for reasonable processor parameters (Figure 2.11). It must be noted that the presented parallelization patterns are commonly used in regular concurrent programming. However, we have been able to beneficially apply them in contexts where the threading overhead would dominate over the parallelization speedup.

Chapter 3 explores the interaction between synchronization and thread manipulation operations and speculative execution. The paper indicates the main difference, which is that speculation is non-linear in presence of multithreading (Figure 3.2): it is no longer true that if an instruction is misspeculated then the instructions that need to be squashed are exactly those that follow it in the program order. As a result, handling misspeculation in a multithreaded processor is hard, and conventional architectures usually perform synchronization only non-speculatively. This solution is unsatisfactory for Inthreads, since thread-related operations are performed very frequently and the latency incurred by delaying synchronization instructions until speculation results in a measurable slowdown.

To allow for execution of thread-related instructions under speculation, we have developed a framework which allows for $O(1)$ detection of the set of squashed instructions. The framework is developed in a general manner and can be applied to any processor that shares a pipeline between several threads (i.e., an SMT-supporting one). The framework requires the processor to identify the pairs of instructions that transfer data between threads. Using this information, it computes timestamp vectors that can be used to determine if a given instruction must be squashed as a result of misspeculation. For efficient implementation, the framework determines the instructions for which the computation of timestamp vectors is absolutely necessary.

The application of the framework to Inthreads is an example of how the Inthreads programming model simplifies the microarchitecture. Since any pair of instructions transferring data between threads must be ordered by a sequence of synchronization and thread management instructions, it is sufficient to consider only the thread-related instructions as communicating ones, reducing the sizes of the relevant hard-
ware structures. Enabling speculative execution speeds up execution of Inthreads-parallelized code by up to 24% (Figure 3.8).

It is important to note that the framework supports thread starting and termination instructions in addition to synchronization. Support of thread starting is straightforward: it can be modeled as communication from the starting instruction to the first instruction of the started thread. However, the speculation model of killing instructions is reverse: while the kill instruction cannot be executed speculatively, it must still be regarded as a communicating one.

Chapter 4 applies Inthreads to increasing the processor’s energy efficiency. One reason for the increased energy consumption is the wasteful and aggressive implementation of modern parallelism-discovery mechanisms. Indeed, the very switch from in-order to out-of-order can be attributed as a reason for more than 60% drop in energy efficiency (Figure 4.2). Our work proposes using a less aggressive and more energy efficient processor implementation, which gains the parallelism, and thus the performance, from Inthreads. To this end, we use an in-order processor pipeline. As Figure 4.12 shows, the Inthreads parallelization achieves about the same speedup as out-of-order organization.

The extensions necessary to implement Inthreads for an in-order pipeline are minimal: the Fetch stage must be extended to handle several threads in a cycle, and a special Wait stage must be added to implement the semantics of synchronization. However, these structures need not be large, as Figures 4.19 and 4.9 show. Furthermore, while the Issue stage must handle instructions of several threads in a cycle, its design is still moderately complex because of the Inthreads programming model, it does not need to perform dependency checks between instructions of different threads.

Our evaluation has shown that Inthreads provides an overall improvement in energy efficiency over both inorder and out-of-order pipelines. As Figure 4.15 shows that under Energy metric, Inthreads over inorder is slightly better than plain inorder, which, in turn, offers a significant energy reduction compared to an out-of-order processor. Under ED metric, inorder pipeline is actually less desirable than out-of-order one, but Inthreads+inorder still keeps the same efficiency advantage. Furthermore, as Figures 4.16 and 4.17 show, our model requires less energy to sustain a given level of IPC as plain in-order, while scaling to much higher IPCs than inorder is capable. The energy benefits of using static parallelism are even more pronounced under ED$^2$ metric, shown in Figure 1.12.

Note that there are architectures that use the same idea to achieve energy efficiency using conventional multithreading [94], however, our architecture offers the same benefit for sequential code.
Chapter 5 discusses the correctness aspects of the implementation of an Inthreads system. It provides a formal definition of the Inthreads programming model and develops frameworks for analyzing correctness of the compiler and the microarchitectural implementation of the processor.

The Inthreads programming model, defined in Section 5.2, is an extension of DRF1 [2]. In the case of Inthreads, the definition applies to the shared registers as well, and the programming model determines that only programs that are free of data races are acceptable.

To analyze correctness of the compiler, we formulated a set of correctness requirements (Section 5.2.3). We proved that if a compiler conforms to the requirements, it will preserve for any compliant program the properties required by the programming model. The compiler’s correctness requirements are applied to the compiler optimizations in Section 5.3 and to the register allocation process in Section 5.4. We characterized the kinds of compiler transformations that need to be adjusted to handle Inth-C code and have shown how to adjust several of those transformations. Using the correctness requirements, it is easy to show that the adjusted transformation algorithms preserve the properties of the programming model.

The part of the compiler that is most affected by Inthreads is the register allocation. While the rest of code optimization need to just preserve the properties of the programming model, the register allocation needs to actively support it. Intuitively, the compiler must ensure that the private variables of different threads use disjoint subsets of the register file, and that the shared variables are always allocated to the same location in different threads. To this end, we present an algorithm that detects the shared variables (correctness of the algorithm also requires the code to be compliant to the programming model), and extend the conventional coloring-based register allocation model to take into account the semantics of shared and private variables. In addition, we analyze the processes of Spilling and Coalescing, which are related to the register allocation, and show that they comply with the compiler correctness requirements.

Register allocation is always successful for serial code, since the spilling algorithm eventually reduces the conflict graph to unconnected state, which is always colorable. However, because of the conflicts between instructions of different threads, the spilling algorithm may never achieve an unconnected graph. In Section 5.4.4 we show that such a scenario is feasible for accurate analysis of reasonable code and outline solutions for ensuring successful allocation.

Evaluation indicates that while Inthreads parallelization puts more pressure on the register file, our register allocation algorithm still works reasonably well. As
Figure 5.20 shows, the original benchmarks are mostly insensitive to the register file size, implying that they don’t make intensive use of all of the registers. Parallelized code, which executes several threads in the same number of registers, is more sensitive, but apart from one benchmark, the increase in pressure is rather small. Indeed, as Figure 5.19 shows, most of the benchmarks are able to reach the maximal speedup at 32 GPRs, indicating that Inthreads should be applicable to existing RISC ISAs.

To support correct implementation of the processor, we formulated a set of correctness requirements for the microarchitecture 5.2.4, and proved that following the requirements is sufficient to comply to the programming model. We use the requirements to show correctness of implementation for both an in-order and an out-of-order pipeline.

The final issue that needs to be handled is the support of function calls under Inthreads parallelization. Function calls, as shown in Figure 5.16, violate the basic assumption of function call, that the caller function is inactive during the execution of the callee. This assumption does not hold when the caller function contains several threads, only one of which runs a subroutine.

One way to address this problem, inlining, can remove function calls altogether, but is not universally applicable: it cannot handle recursion, system and library calls. Still, there are circumstances, listed in Section 5.6, where performing function calls from within Inthreads-parallelized code is necessary to enable parallelization. To support function calls, we developed a mechanism for suspending thread execution for the duration of a function call. Similarly to the rest of the Inthreads programming model, the suspend mechanism requires the software collaboration to operate.

The hardware provides the ability to switch from multithreaded execution to single-threaded one and back. During the switching, it must bring the processor to a known good state, including waiting the pipelines of all the threads to empty and purging instructions that are stuck on synchronization. Figure 5.17 shows the processor states involved in the switching.

The software is responsible for saving and restoring the execution context of the threads, in addition to the standard calling sequence. This includes saving the PC registers of the threads and the current state of all the condition variables. In addition, the function return process involves resetting and restoring the TID of the current thread. This is also performed by the program, as described in Section 5.6.1. The Inth-C compiler provides automatic support for protecting the function calls: each call performed within a parallelized code section is enveloped in a suspend/restore sequence. However, the programmer can specify larger blocks of protected code to reduce the number of suspends necessary.
Bibliography


לنزل את זהות ולהושב על האינטראקציה בין החוטים. כל נסך ביניהם הוא מתפלшим ו
בזכותם ערך התוכנות של Java. מודל התוכנות המורכב הווהי של Java, המודר תלמידי
התרבות של מענה ספציפי, ואנו מסיימת אותו, או שהוא בוחר לחובב
ולא יכול למסתובב בגשתה, מבית וחברת יזום. שהבטיח שיובא לאופרטיבית. 3. מודר במדחיבות של ביצוע התוכנות של הצל אתแนะה לשון קים
מסיימם. אתון השון חכם דמדח את התוכנות של Java, עם מודלי أفريقي, וממנים שוהי קים
측当たり, שאם מתווה את דמדח התוכנות של Java, עם מודלי أفريיתיבי, הננו מדויק וחד משמעי.
יחד עם זאת, הם קשים ביותר לחשיבה ולאנליזה גם לממנים למערכת גם לממנים.
העבודה שלנו מפתחת תיאור זהcimiento זה, בנה ברמה של ביצוע התכנית עצמה
בennent נהיה במונחים של ביצוע התכנית עצמה וללא קשר למסלים מסוימים.
אנחנו השווינו את מודל התוכנות של Java עם מודלים אחרים,
ומצאנו ש🇨 (!$ ויקים
הגילויים אלה מדגים את הכוח של מודלי הסופיות
כ安检ון משון, ו nội לדרישת ה-
 diferença ישירות מהמודל שלנו
ולעומת זאת, אם-definition determinacy
שהדבר יכול להימנע מכיוון שה-
Coherency נובע ישירות מהמודל שלנו.
לשם, הארכיטקטורה המרובה התוכנות המופרות, בשתיו של מיני.Margin שהמקבל
בתוכנות, המ الداخل את פוריות בנים הצוותים של מיני Margin במקבל
מקבל במחזור, ביצועים. בצמם, מこそות להשלישיות בציור, בודקים רמה כמ MIDI Margin
לזה יעןepy après ארבעה לתוך מתחומ התוכנות ממחזור, ביצוע במקלדת התוכנות של॥ וכמק
סידורי בלבד.


Inthreads, a new level of parallelism

In this work, we address one of the most important problems of modern computer architectures, which is the expression of parallelism in programs. The work proposes a new level of parallelism that is located between the existing levels, and essentially allows the execution of multiple threads in programs that are sequentially ordered in existing architectures.

The proposed architecture, Inthreads, tries to utilize a low to moderate level of parallelism that is not utilized by instruction-level parallelism (ILP) engines, which are commonly used in out-of-order execution, and are limited to the level of ILP engines operating at the level of threads. Despite the engines reaching almost full utilization of the parallelism available at relevant levels, their field of operation is limited.

On the one hand, the complexity of implementation and the interdependencies between instructions prevent out-of-order execution from looking far ahead in a sequence of instructions.

On the other hand, time constraints related to the execution of synchronization and thread management prevent thread-based architectures from handling small problems. The Inthreads architecture solves this complexity through a simple thread mechanism.

By a certain flexibility, it reduces the complexity of managing threads to a minimum similar to the complexity of managing individual instructions in regular architectures.

In order to do this, Inthreads defines a programming model based on a fixed number of threads that can be executed simultaneously, and on the sharing of architectural registers.

The programming model requires active participation of both hardware and software.

Under this model, the threads work together to divide the resources of the processor, including the contexts and architectural registers.

In terms of software, the architecture requires a deep change in the compilation process. Due to the close sharing of resources between the threads, the compiler must work together with the code of all threads that are executed simultaneously, and this requires adaptation of the intermediate representation and many of the compiler's algorithms.

For this purpose, we define a set of requirements that guarantee that the compiler maintains the characteristics required by the programming model.

With the help of this set of requirements, we examined the optimizations of the compiler and showed how to adapt them to a compiled program using the programming model.
המחק נשע עלهجה פרופ', אסף שוסטר בפקולטה למדעי המחשב.

הכרת תודה

ברצוני להודות לפרופ' אסף שוסטר על עידוד קבוע ותמיכה בלתי מתפשרת נולד והומצאת עמוד הקמתו של הפקולטה למדעי המחשב. הוא הביא את האswickים של המתים לקאתה בלתי מתפשרת. הוא הביא את האswickים של המתים לקאתה בלתי מתแทงบอลים על חברה שבועית של אסף ומסור锜ים.

בנוסף, כל מה שיאפשרו ידוע על אסף ומוריץ, למדתי מהיים ותומים של אסף משוגעים על אסף משוגעים.

אני מודה לדר' אבי מנדלסון על האינטואיציות המדהימות והнныеיות של אסף.

מספר אנשים הלכו איתי את הדרך ועזרו לקצרה משמעותית. אני מודה לסרגיי פוליאקוב, גרגורי שקולובר וולדימיר זדורנוב על כל המחשבה והעבודה הקשה שהם השקיעו בכלים ובראיונות שקידמו את העבודה הזאת.

אני מודה לדר' יבגני גברילוביץ' על דברים רבים מדי כדי להזכירם, אך במיוחד על "דרך הגאבר" של התייחסות לדברים שלו.

אנשים רבים עזרו לי בדיונים, עצות ותמיכה. אני מודה לכולם: סיון ברקוביץ', אלכסנדר ברנגולץ, דוד ברנשטיין, גנאדי בריוזקין, אדוארד בורטניקוב, ויטה בורטניקוב, מקסים קובגן, ודיב מקרויקס, בילה מנדלסון, אנריק מורמל, ואילס רימר, מתאו ולרו, ויטלי סורז, סקיי, טטיאנה סורז, סקיי ואייל זקס.

אני לא יכול להודות מספיק/varav האהובה, יוליה, על אהבתה ותמיכתה. היא她在 שידי, עם סבלנות ושקוטה, ובהתרסה בשתייה של אסף ומסור锜ים.

אני מודה לטיבי מת באופן דר基本的に מה[maxnך עזיב, והקרזח של הולכות ולכתחוד בהיותם מיתסולם בהיותם התרבותית של התחדשות.

לבסוף אני מודה לקצרה של כל ההתרסה ולכתחוד של כל החוקרים והחוקיות המขวาים.

אני מודה ל㈑ご利用 והקצאת הכספים הנדיבות ואתה פעילות המיתסולם בהיותם התרבותית של התחדשות.
אורכי-קטורה ומודל תכנות למקבול
בגרעיניות נמוכה מאוד

חיבור על מחקר
לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה
אלכס גונטמכר
הוגש לסנט הטכניון - מכון טכנולוגי לישראל
סיוון תשס"זחיפהמאי 2007
ארכיטקטורות ומודל תכונת למקבלי
בגרעיניות נמוכה מאוז

אלכס גונטマー