Explicitly-Parallel Code Compilation
Methods for Shared-Context Architecture

Research Thesis

Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Science

Gregory Shklover

Submitted to the Senate of
the Technion – Israel Institute of Technology

SIVAN, 5766  HAIFA  June, 2006
Acknowledgment

The research thesis was done under the supervision of Prof. Assaf Schuster in the faculty of Computer Science, Technion, Israel.

I would like to express my greatest gratitude to Alex Gontmakher for his infinite patience and professional guidance.
# Table of Contents

1 Introduction.................................................................................................................. 3  
  1.1 Contributions........................................................................................................ 3  

2 Inthreads Architecture.................................................................................................. 4  
  2.1 Motivation............................................................................................................... 4  
  2.2 Inthreads Architecture Programming Model....................................................... 5  
    2.2.1 Inthreads ISA................................................................................................... 5  
    2.2.2 Inthreads Parallel Execution Model................................................................. 6  
  2.3 Inthreads-C Syntax.................................................................................................. 7  

3 Explicitly-Parallel Code Compilation............................................................................ 10  
  3.1 Compilation Flow.................................................................................................... 10  
  3.2 Compilation Correctness........................................................................................ 10  
  3.3 Internal Representation.......................................................................................... 11  
    3.3.1 Concurrent Control Flow Graph........................................................................ 11  
    3.3.2 Concurrent Static Single Assignment............................................................... 13  
  3.4 Analysis Techniques.............................................................................................. 13  
    3.4.1 Shared Variable Analysis.................................................................................. 13  
    3.4.2 Inthreads Program Analysis............................................................................. 15  
    3.4.3 Parallel Data Flow Analysis............................................................................ 18  
  3.5 Optimizations......................................................................................................... 20  
    3.5.1 Dead Code Elimination..................................................................................... 21  
    3.5.2 Constant and Copy Propagation...................................................................... 22  
    3.5.3 Common Sub-Expression Elimination............................................................. 23  
    3.5.4 Loop Invariant Code Motion........................................................................... 24  

4 Register Allocation...................................................................................................... 25  
  4.1 Register Allocation for Sequential Code................................................................. 26  
  4.2 Register Allocation for Concurrently-Executing Code............................................. 27  
    4.2.1 Register File Partitioning................................................................................. 28  
  4.3 Register Allocation by Graph Coloring................................................................. 29  
    4.3.1 Background...................................................................................................... 29  
    4.3.2 Parallel Interference Graph.............................................................................. 31  
    4.3.3 Coalescing....................................................................................................... 34  
    4.3.4 Spill Code Generation...................................................................................... 35  
      4.3.4.1 Spill Everywhere......................................................................................... 35  
      4.3.4.2 Interference Region Spilling................................................................. 38  
      4.3.4.3 Short Live Ranges.................................................................................... 41  
      4.3.4.4 Evaluating Spill Cost Heuristic.............................................................. 42  
    4.3.5 Complexity Considerations............................................................................. 48  
  4.4 Optimal Register Allocation.................................................................................... 49  
  4.5 Related Work.......................................................................................................... 51  

5 Implementation Details............................................................................................... 53  

6 Conclusions................................................................................................................ 55  

Bibliography.................................................................................................................... 56
Illustration Index

Figure 1: Parallelization granularity comparison.................................................................4
Figure 2: Inthreads-C syntax example....................................................................................8
Figure 3: Inthreads compilation model..................................................................................10
Figure 4: Parallel code example............................................................................................11
Figure 5: Concurrent Control Flow Graph...............................................................................13
Figure 6: Shared variable example.........................................................................................14
Figure 7: Algorithm to identify shared variables.................................................................14
Figure 8: Delays in a critical cycle........................................................................................18
Figure 9: Generated and killed definitions in a data flow problem.........................................19
Figure 10: Using sync. edges for data flow analysis..............................................................19
Figure 11: Copy propagation correctness...............................................................................22
Figure 12: Common sub-expression elimination.................................................................23
Figure 13: Example of an abstract representation with virtual registers...........................25
Figure 14: Example of register file partitioning.................................................................29
Figure 15: Graph coloring algorithm flow............................................................................30
Figure 16: Live ranges............................................................................................................31
Figure 17: Using synchronization for finding live ranges....................................................32
Figure 18: Algorithm to collect intra-thread live range conflicts.........................................32
Figure 19: Concurrent execution examples...........................................................................33
Figure 20: Interference graph example................................................................................34
Figure 21: Register coalescing in parallel code.................................................................34
Figure 22: Example of spill code generation........................................................................37
Figure 23: Spilling read-only shared variables.................................................................38
Figure 24: Sequential interference region spilling............................................................39
Figure 25: Race condition due to shared variable reload......................................................40
Figure 26: Selective interference region spilling.................................................................40
Figure 27: Data race on reloading a read-only variable.......................................................41
Figure 28: Cross-interference of short live ranges..............................................................42
Figure 29: Interference graph where greedy algorithm fails...............................................42
Figure 30: Genetic programming progress.........................................................................45
Figure 31: Spill cost of expression length...........................................................................45
Figure 32: GP expression vs. default..................................................................................46
Figure 33: Spill cost for different parallelization degree and number of registers.................47
Figure 34: Dynamic performance degradation...................................................................47
Figure 35: Number of evaluated nodes as function of the graph size..................................49

Index of Tables

Table 1: Allocation cost as function of register pressure.......................................................44
Table 2: Best expressions of specified length.......................................................................46
Table 3: Interference graph statistics..................................................................................48
Abstract

In this work we investigate compilation methods for explicitly parallel code targeted at Inthreads – a lightweight, shared-context multi-threaded architecture.

The Inthreads architecture provides the means for expressing parallelism at a very low granularity level. Contrary to most other architectures, Inthreads executes all the threads in a shared architectural context: the threads share not only the memory, but also the register file. It also provides a set of basic synchronization primitives and defines strict rules for shared resource accesses.

We use a two-stage compilation flow where an automatic parallelizer or a programmer provides an explicitly-parallel source code, that is then compiled into machine code by an Inthreads compiler. The compiler must be capable of understanding the semantics of the parallel code and applying optimization transformations while preserving code correctness.

To preserve correctness of the compiled parallel code during the compilation process we modify the traditional analysis and optimization algorithms. We also address register allocation which is an Inthreads-specific compilation challenge.

Shared register file, as provided by Inthreads, is rarely available in other parallel processing frameworks. In this work, we investigate the register allocation for parallel threads and show that graph coloring register allocation model can be natively extended to handle register allocation for parallel threads. We also present empirical results on the quality and complexity of this algorithm.
Index of Common Terms

**close references** - two variable references are close if no other variable dies between them. (page 36)

**concurrent** - two events are said to be concurrent if they are not ordered by happened before relation. (page 6)

**data flow analysis** - compile-time analysis technique for deducing run-time information dependencies between program statements. (page 18)

**data race free** - a program is said to be data race free if there is a happens-before-1 relation between any pair of conflicting events in the program. (page 7)

**definition** - a write access to a variable. (page 18)

**happened before** - a relative order between events in a parallel execution system. (page 6)

**happens-before-1** - a relative order between events in a parallel execution system induced by program order and synchronization events. (page 7)

**interference** - two live ranges are said to interfere if they can not be allocated with the same machine register. (page 31)

**live variable** - a variable is live at a location if it is used in the following control flow and not assigned before the use. (page )

**load** - read memory access. (page 35)

**parallel threads** - the set of parallel threads is determined by the explicitly-parallel source code semantics. (page 7)

**program order** - a relative order between events prescribed by the program text. (page 6)

**live range** - an independent part of a virtual register. (page 30)

**SMT** - simultaneous multithreading

**spill code** - memory load and store instructions added during register allocation to place values on stack. (page 25)

**store** - write memory access. (page 35)

**use** - a read access to a variable. (page 18)

**variable death** - a variable use after which the variable is no longer live. (page 36)

**virtual register** - represents a scalar variable or a compiler-generated temporary in internal code representation. (page 25)
1 Introduction

Intrinsic parallelism is one of the major sources for performance improvement in computer architecture. The parallelism exists at many granularity levels. At the coarsest granularity, a task can be divided between different machines, while the finest granularity involves parallel execution of individual instructions. Since the amount of parallelism at each granularity level is inherently limited, it is important to use a combined solution to fully realize parallelism potential.

*Inthreads Architecture* (Inthreads) targets the medium and low granularity levels and complements solutions for instruction-level (ILP) and thread or process-level parallelism (SMT/SMP). The architecture uses lightweight threads to represent concurrent execution flows. The threads share the execution context almost completely, down to the architectural register file. Only some registers are allocated per-thread, such as the program counter (PC).

Inthreads code generation is mostly intended to be carried out by a parallelizing compiler. We divide the code generation into two phases: *parallelizing optimization* (producing explicitly-parallel intermediate language) and *explicitly parallel code compilation*. Both present interesting challenges and have been a target of academic research during the last years. In this work we concentrate on the explicitly-parallel code compilation, where Inthreads architecture introduces a set of new problems.

1.1 Contributions

We explore the methods for explicitly-parallel code compilation in shared-context architecture (*Inthreads*). The main contributions of this work are in the following areas:

- We extend the traditional graph coloring register allocation algorithm to cover register allocation for concurrently executing threads. We consider correctness of each of the algorithm steps: building an interference graph, coalescing and spill code generation. We also empirically evaluate the quality and performance of this algorithm under the new constraints introduced by the parallel architecture.
- We define a programming model that supports correct compilation for explicitly-parallel programs targeted at Inthreads architecture. The model is constructed by combining correctness requirements for the source program and the compilation process.
2 Inthreads Architecture

2.1 Motivation

Although the number of transistors being packed into a single processor constantly grows, it is evident that instruction-level parallelism can not fully utilize this computational strength. On the other hand, power and manufacturing process limitations prove that frequency scaling can no longer be the sole answer to computational performance. In an effort to overcome those limitations, recent architectures attempt to boost the utilization of a single-die processor by running multiple processes or threads on a single physical processor [28]. Exposing computational parallelism was also one of the major research topics in computer architectures during the last years.

*Inthreads architecture* targets cases where instruction-level parallelism (ILP) is insufficient to deal with complicated control flows or memory latency, while thread-level parallelism remains inappropriate due to its relatively high overhead. While ILP exposes inherent parallelism along a single execution path, *Inthreads architecture* allows multiple non-speculative control flows to be executed simultaneously. Unlike traditional SMT architectures, Inthreads runs the threads in an almost completely shared execution context of a single processor.

Efficient implementation of this architecture can lower the latency of the thread-related operations (for example by means of speculative execution of synchronization operations). Such implementation can effectively expose more optimization opportunities to traditional parallelizing techniques, otherwise inapplicable due to thread management overhead.

Figure 1 presents a speedup comparison obtained from parallelizing a simple loop with two threads using

![Figure 1: Parallelization granularity comparison](image-url)
four different methods: Inthreads, pthreads (posix threads) with busy-wait synchronization, pthreads with synchronization primitives and dynamically created pthreads. The X axis corresponds to the number of loop iterations performed by each thread. The results indicate that Inthreads parallelization reaches break-even after just two loop iterations, while pthreads requires almost 3 orders of magnitude more.

Since Inthreads architecture requires very little OS support, it could safely coexist with traditional thread-level parallelism. One could consider, for example, a super-scalar SMT CPU running two OS-level threads, each running several *inthreads*.

2.2 Inthreads Architecture Programming Model

2.2.1 Inthreads ISA

*Inthreads* uses a lightweight architecture-level representation for parallel threads. An *inthread* is merely represented by a PC register that indicates the thread execution point. Other registers (with exception of some special purpose registers) are shared by all parallel threads. It also supports architecture-level synchronization primitives with binary semaphores and corresponding *wait* and *set* instructions.

Such representation is different from traditional SMT methods that usually implement parallel constructs by library routines, system calls, and use shared memory for cross-thread communication and synchronization. Inthreads require minimal system-level thread management. Thread scheduling is performed by the CPU as it fetches and executes instructions for all of the executing threads. The operating system is only required to stop all the threads during a context switch and store small amount of additional registers.

Below is the set of threads-related instructions defined by the Inthreads:

- **inth.start <id>,<addr>** Starts a new thread at specified code location.
- **inth.halt** Terminates the current thread. This instruction executes synchronously guaranteeing completion of the preceding instructions.
- **inth.kill <id>** Kills specified thread. Unlike inth.halt this instruction is asynchronous – there is no guarantee on the exact point where the thread will be terminated.
- **inth.clear <num>** Clears specified binary semaphore state.
- **inth.wait <num>** Suspends thread execution until the semaphore is set. The value of the semaphore is automatically cleared when the thread continues.
- **inth.set <num>** 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 <addr>** Suspends running threads and saves the CPU context to the specified memory location. Such functionality can be used by the operating system.
or by the compiler to suspend the parallel threads.

```
inth.resume <addr>
```

Restores the context from the specified memory location.

The architecture has a fixed number of parallel threads. The actual number of parallel threads is usually known at compilation time. The fixed number of threads does not pose a problem as number of parallel threads is effectively limited by the use of the shared register file.

The register file sharing has some drawbacks. Most commonly-accepted function call conventions assume a single active function call frame and store inactive call frames on the stack. Calling a function while other threads are active would violate above assumption: the function, unaware of the running threads, could attempt to use the same registers creating a data race.

In many cases this problem can be mitigated by function inlining. For cases where inlining is inapplicable, the architecture provides the instructions to suspend and resume threads in the active stack frame. This mechanism preserves the same “single active call frame” assumption: the callee function can assume the usual call conventions and can safely access the registers of the suspended threads.

### 2.2.2 Inthreads Parallel Execution Model

In the following, we give definitions to help us formally define the parallel execution model provided by the Inthreads.

A parallel system can be modeled by a set of processes, each consisting of a sequence of events. The processes communicate by exchanging messages — those may represent the actual messages in a distributed system or memory accesses in a shared memory environment. The *happened before* relation [26], denoted by \( \rightarrow \), is used to define a partial order between the events that occur in the system and to resolve the inter-process synchronization problems. For two events \( a \) and \( b \), \( \rightarrow \) is defined as follows: if both events belong to the same process and \( a \) is issued before \( b \), then \( a \rightarrow b \); if \( a \) is a message sending event and \( b \) is the receiving event of the same message, then \( a \rightarrow b \); the relation is transitive: \( a \rightarrow b \) & \( b \rightarrow c \Rightarrow a \rightarrow c \). Two events are said to be *concurrent* if they are not ordered by \( \rightarrow \) relation. In practice (and in this work) we usually operate on ordered or concurrent blocks of instructions rather than individual events.

The execution of the parallel processes (threads) is often associated with *interleaving semantics*: the result of the execution is modeled as if the instructions of all the threads where executed in some global sequential order. The relative order of instruction for each specific thread in this sequence is prescribed by the original program text and is called the *program order* [37]. Execution orderings that follow interleaving semantics are also defined to be *sequentially consistent*.

Most implementations provide relaxed execution models where the term *performed with respect to* is used to define the observed order of events. A *read* accesses issued by process \( P_1 \) is said to be *performed with respect to* process \( P_2 \) if issuing a *write* by \( P_2 \) to the same address can not affect the value read by \( P_1 \).
Similarly, a write access by $P_1$ is performed with respect to process $P_2$ if a read by $P_2$ will return the value written by $P_1$ (or a later write). An access is performed when it is performed with respect to all processes.

Two accesses are said to conflict if both access the same location and at least one modifies it [37]. Inthreads architecture offers a Release Consistency (RC) execution model [12]. RC categorizes different types of conflicting accesses in a parallel program: synchronization (further partitioned into release and acquire), competing and non-competing. A release is always associated with a write synchronization access, while an acquire is always associated with a read synchronization access.

A system that conforms to RC must satisfy the following execution conditions: (a) before a read/write accesses is allowed to perform with respect to any other process, all previous acquire accesses must be performed; (b) before a release is allowed to perform with respect to any other process, all previous read/write accesses must be performed; (c) competing and synchronization accesses are sequentially-consistent with respect to one another.

A program is said to be data-race-free if for any pair of conflicting accesses, there is a happens-before-1 relation [1] between them for any interleaving execution of the program. This relation is induced by the transitive closer of the program order and the order between the synchronization events of the form release(x)$\rightarrow$acquire(x) where the acquire(x) event returns the value written by the release(x). Execution of such programs on an RC model is also equivalent to sequentially-consistent execution [12].

The Inthreads architecture explicitly defines the set of synchronization accesses and their semantics with regard to shared memory and register file accesses: inth.start, inth.halt, inth.set and inth.resume instructions have release semantics, while inth.wait and inth.suspend have acquire semantics.

It also requires the programs to be data-race-free, thus providing the programmer with sequentially-consistent execution environment. Explicit synchronization must be used to coordinate data sharing between parallel inthreads.

As a consequence of the race-free requirement, the rest of the shared accesses in an Inthreads program are categorized as non-competing [12] by definition. This eliminates the possibility of some parallelization patterns, such as busy-wait synchronization, but allows for a significantly simpler hardware implementation [13].

2.3 Inthreads-C Syntax

Inthreads-C is an extension to the C language syntax that adds support for Inthreads parallelization constructs. It is used as an intermediate language between a programmer or the parallelizing front-end and the explicitly-parallel code compilation back-end.

Figure 2 shows an example of Inthreads-C code. Each thread is enclosed in a syntactical block with a #pragma inthread header. Each such block contains the code of a single thread and can not be nested inside
another #pragma inthreads block. Each of the Inthreads instructions has a corresponding representation in the Inthreads-C syntax:

- \texttt{INTH\_CLEAR}(1);  
- \texttt{INTH\_START}(1,\texttt{start2});  
- \texttt{#pragma inthread}  
  \{  
  int i;  
  for (i = 0; i < N; i+= 2) { sum1 += buff[i];}  
  \texttt{INTH\_WAIT}(1);  
  \}  
  return sum1 + sum2;  
  \texttt{start2:}  
  \texttt{#pragma inthread}  
  \{  
  int i;  
  for (i = 1; i < N; i+= 2) { sum2 += buff[i];}  
  \texttt{INTH\_SET}(1);  
  \texttt{INTH\_HALT}();  
  \}  

\texttt{a) original code} \hspace{1cm} \texttt{b) parallelized (Inthreads-C) code}

\hspace{1cm}

Figure 2: Inthreads-C syntax example

\hspace{1cm}

\begin{verbatim}
int i;
for (i = 0; i < N; i++) {
  sum += buff[i];
}
return sum;
\end{verbatim}

\begin{verbatim}
INTH\_CLEAR(1);
INTH\_START(1,\texttt{start2});
\texttt{#pragma inthread}  
\{  
  int i;  
  for (i = 0; i < N; i+= 2) { sum1 += buff[i];}  
  \texttt{INTH\_WAIT}(1);  
  \}  
  return sum1 + sum2;  
  \texttt{start2:}  
  \texttt{#pragma inthread}  
  \{  
  int i;  
  for (i = 1; i < N; i+= 2) { sum2 += buff[i];}  
  \texttt{INTH\_SET}(1);  
  \texttt{INTH\_HALT}();  
  \}
\end{verbatim}

Besides the syntactical representation, the above commands bear additional compile-time semantics. Merely mapping Inthreads instructions with in-line assembler would be insufficient. As we will show in Section 3.2, to perform correct code transformations, the compiler must be aware of the semantics of the parallel code.

\textit{Inthreads-C} does not require shared variables to be syntactically marked. Variable visibility scope follows regular language rules: a variable declaration visible by two threads may be used for communication. Working at such a low abstraction level, the Inthreads compiler is capable of automatically identifying the set of shared scalar variables.

A correct \textit{Inthreads-C} program has to make sure that only a single thread is alive when execution reaches a point outside the block marked with \texttt{#pragma inthread}. Other language extensions, such as parallel Fortran, use implicit synchronization to merge the threads. Working at a lower-level of abstraction, the Inthreads-C uses explicit synchronization to fulfill this requirement.

The Inthreads-C extension can be compared to the widely accepted OpenMP parallel program interface.
Both use #pragma statements for defining explicit parallelism: the Inthreads-C #pragma inthread is similar to the #pragma parallel sections OpenMP construct. The specifications differ at the level of the abstraction. OpenMP generally operates on a higher abstraction level: thread management code is automatically generated by the compiler. It is also intended to enable both sequential (when pragmas are ignored) and parallel (when OpenMP pragmas are honored) execution of the program. Inthreads-C, on the other hand, exposes low-level Inthreads operations: parallel code sections and thread management operations are specified explicitly. Inthreads-C also cannot be executed sequentially.
3 Explicitly-Parallel Code Compilation

3.1 Compilation Flow

We divide the compilation process into two steps: parallelization front-end (performed by a parallelizing compiler or a programmer) and explicitly-parallel code compilation back-end (Figure 3).

In this work we focus on the back-end stage of the flow: given explicitly parallel code, written by a programmer or produced by a parallelizing compiler, we need to compile it into target architecture machine code (Inthreads).

We use the following observations regarding the compilation process in general:

- A compiler uses internal representation – an abstract representation of compiled code. The type and level of the abstraction is usually tightly linked to the types of analysis and optimization algorithms used by the compiler.

- A compiler uses a set of analysis techniques to derive information about run-time code behavior at compilation time. This information is later used during code transformations to preserve intended run-time behavior of the program.

- Compilation process usually consists of transformations, during which original code representation is modified (optimized) or transformed into a different representation. Those transformations are usually guided by the information provided by code analysis. In most cases, transformations are applied in safe or conservative manner – a transformation must not modify observable program behavior.

Each of those compilation aspects is important for correct processing of parallel code and those were already considered in the past for different parallel frameworks [38,27,20,19,37,25]. Inthreads-specific features, such as shared register file, add new challenges to the parallel code compilation.

3.2 Compilation Correctness

Moving from sequential to parallel code compilation has fundamental influence on the basic compilation concepts that when applied naïvely result in an incorrect compilation.
Consider the code sample given in Figure 4 (assuming that listed threads execute in parallel). Applying traditional \textit{dead code elimination} to the above code could, for example, identify the assignment in line 10 as \textit{dead code} - code that produces a value never used by subsequent instructions. The compiler could then erroneously remove the assignment to \( a \), changing the program behavior.

Dead code elimination algorithm, as many other optimizations, is based on \textit{data flow} information usually provided by a generic data flow analysis. This fundamental analysis technique, as well as some other methods used for sequential code, can not be trivially applied to parallel code. Code motion optimizations are another example: trivially applied to a sequentially consistent program, they may violate its consistency [37].

Synchronization constructs and memory consistency models are yet another field where sequential compilers had little interest prior to parallel code compilation. For example, instruction scheduling pass could violate the RC synchronization semantics if it reorders lines 4 and 5.

An Inthreads-specific correctness aspect is related to the fact that threads execute in the same context: memory is not the only, and in fact not the primary, shared resource. The processor's architectural register file serves all parallel threads for both thread-local and thread-shared data. The compiler is responsible for performing register allocation for threads in a way that preserves code semantics.

Early compilers that supported explicitly-parallel languages used to disable optimization entirely on the parallel sections of the code in order to preserve original code semantics. But, as studies in this area progressed, generic analysis techniques [37,19] as well as methods for individual optimization steps [20] were proposed to extend common optimization algorithms to parallel code compilation. As part of our work, mostly addressing the register allocation, it was also required to perform the necessary modifications to the internal representation, analysis and optimization algorithms to preserve compilation correctness. The remaining part of the section discusses our modifications in more detail.

3.3 Internal Representation

During the compilation process a source program usually undergoes transformations between several different representations. Each representation provides the compiler with an abstraction where generic optimization algorithms can be applied to transform it to semantically-equivalent, but more efficient form (where “efficient” is determined by the optimization objective which could be performance, code size, or both).

3.3.1 Concurrent Control Flow Graph
Most modern compilers have internal representation of the compiled code in a form of control flow graph (CFG) [30]. A CFG consists of basic blocks connected by control edges, where basic blocks represent the operations performed by the program and the control edges represent possible directions of the execution flow. The implementation of the optimization algorithms used by the compiler usually determines the level of abstraction used in the representation: an iterative data flow algorithm can work on a low-level abstraction, where blocks in the graph comprise the actual instructions; while a structural approach to data flow may use higher abstraction level and have blocks that represent specific language constructs (such as while-loop blocks and if-then-else block etc.)

Concurrent Control Flow Graph (CCFG) is an extension of the control flow graph representation that accommodates the additional information on the parallel execution and synchronization.

The Inthreads compiler uses a form of CCFG that reflects the parallel structure of an Inthreads-C program. We use extended definition of a basic block. A basic block is a sequence of instructions, where control may enter only at the beginning and leave only at the end, without the possibility of branching other than at the end. It also has properties specific to parallel code representation:

- A condition wait (inth.wait) is always the first instruction of a basic block. Such partitioning is required for proper representation of synchronization edges described later.
- A condition set (inth.set) is always the last instruction of a block.
- The inth.start instruction is always the last instruction of a basic block with two successors, but unlike other branching commands, each edge has 100% branching probability.
- inth.halt is the last instruction of a basic block without control flow out of the block.

The CCFG has two special edge types in addition to traditional sequential control flow edges: parallel control flow edges and synchronization edges. Parallel control flow edges represent parallel execution of both paths following an execution of an INTH_START() and are mostly handled as regular control flow edges. Synchronization edges connect INTH_SET() instructions with corresponding INTH_WAIT() instructions in parallel threads and are used to represent possible synchronization dependencies between them.

Similarly to other C-language jump rules, there should be no direct jumps into the parallel code: a thread code may be entered only by following the edges of a block ending with an INTH_START() instruction.

Figure 5 shows a CCFG that can represent the example from Figure 4. Two parallel edges leave the INTH_START() block to represent parallel threads. Thread #2 terminates with an INTH_HALT() block. There is also a synchronization edge connecting the corresponding INTH_SET() and INTH_WAIT() instructions.

A number of CCFG variants were proposed in the past [27,38,39]. Our implementation utilizes some of the common features of a concurrent control flow graph with several structural differences dictated by the
Inthreads code compilation requirements.

Our model uses explicit thread management. Other implementations, such as those proposed for Parallel Fortran `cobegin-coend` [38], generate an implicit edge connecting the threads at the end of the parallel code section. Since Inthreads-C requires that only a single thread is alive when control reaches the code outside `inthread-marked` block, the parallel code uses explicit synchronization to synchronize thread termination (or an explicit INTH_KILL()). This synchronization edge provides a merge point similar to those used for representing Fortran parallel sections.

In [27] Lee uses CCFG structure with conflict edges that connect references to shared variables in parallel threads and can be used during delay set analysis [37] for identifying critical cycles. We use a different analysis technique (described in detail in Section 3.4) that does not require conflict edges.

In [38] the CCFG representation uses non-uniform blocks to create a hierarchical structure. This structure is required to support nested parallel sections in parallel Fortran. Since Inthreads-C does not allow nesting of a `#pragma inthread` block inside another `#pragma inthread` block, we use a uniform (flat) CCFG structure.

3.3.2 Concurrent Static Single Assignment

Static Single Assignment (SSA) [30] is an alternative code representation based on control flow graph that provides ground for efficient implementation of optimization algorithms. The main property of this representation is that for each variable use there is a single variable assignment that reaches that use point.

Due to its properties, SSA proved to be a popular representation for various optimization algorithms. Although this representation can be extended to handle parallel code compilation (Concurrent Static Single Assignment in [39,27]) we do not describe it in this work. The optimizations we discuss are based on the CFG representation, but are equally applicable and have corresponding implementations with SSA form.

3.4 Analysis Techniques

3.4.1 Shared Variable Analysis

In this work, we focus on scalar shared variables analysis, but similar methods could be used to detect shared memory locations based on memory aliasing analysis.

The following definition of a live variable will be required for our later definitions:

A variable \( v \) is live at execution point \( e \) if there is a control path from \( e \) to some use of \( v \) and there is no assignment to \( v \) on that path.
We use the live variable formulation and the term concurrent defined in Section 2.2.2 to define a shared variable:

Variable \( v \) is shared if it is modified at an execution point \( e_1 \) and live at an execution point \( e_2 \) concurrent with \( e_1 \).

This definition is slightly different from the common one, where a variable \( v \) is considered to be shared if there are conflicting accesses to \( v \) issued by parallel threads. The Inthreads compiler requires our extended definition to recognize the data flow between a thread and the sequential code executed after the thread termination. The second definition doesn't identify variables written by a thread and accessed after the thread is terminated as being shared.

The difference can be demonstrated using the code example in Figure 6. Although variable \( a \) is modified by one of the threads, it is not used by the other thread and,

```
INTH_START(start2);
#pragma inthread
{
    INTH_WAIT(2);
    print(a);
}
```

```
start2:
#pragma inthread
{
    a = ...;
    INTH_SET(2);
    INTH_HALT();
}
```

*Figure 6: Shared variable example*

---

```
procedure IdentifySharedVars(CFG graph)
    ComputeLiveInfo(graph)
    SharedVars = {}
    foreach block in graph do
        if (block is INTH_START()) do
            \(<\text{defined}_1, \text{live}_1> = \text{CollectDefsAndLive(block.successors[0])};
            \(<\text{defined}_2, \text{live}_2> = \text{CollectDefsAndLive(block.successors[1])};
            SharedVars U= \text{defined}_1 \cap (\text{defined}_1 U \text{live}_2);
            SharedVars U= \text{defined}_2 \cap (\text{defined}_2 U \text{live}_1);
            done
        done
    end

procedure CollectDefsAndLive(Block block)
    DefVars = {}
    LiveVars = {}
    foreach insn in block do
        DefVars U= insn.defined;
        LiveVars U= insn.live;
        done
    Alread Visited U= \{block\}
    foreach succ in block.successors do
        if (EdgeLeavesThread(block, succ)) continue;
        if (block in Alread Visited) continue;
        \(<\text{DefVars}, \text{LiveVars}> U= \text{CollectDefsAndLive(succ)};
        done
    return \(<\text{DefVars}, \text{LiveVars}>\)
end
```

*Figure 7: Algorithm to identify shared variables*
thus, doesn't qualify under the second definition. Without the assignment to \( a \) being marked as communicating, the compiler may move it to after the following INTH_SET() instruction violating intended program behavior.

Figure 7 lists the pseudo-code for the algorithm used by the Inthreads compiler to identify scalar shared variables. The algorithm considers each statement of a thread to be concurrent to all the statements of a parallel thread and finds a conservative set of shared variables that match our definition.

First, it computes sequential liveness information according to the above definition and denoted by ComputeLiveInfo() [30]. Then, for each INTH_START() block the algorithm collects the defined and the live values from each of the code segments pointed by the INTH_START() branch. Values defined by one thread and live or defined by a parallel thread are then marked as \textit{shared}.

For the purpose of this discussion, the algorithm in Figure 7 presents a simple implementation: each pair of parallel threads will be visited once, but a specific thread may be visited more than once (\( O(p) \) times, where \( p \) is the number of parallel threads). Given the fixed and relatively small number of parallel threads this doesn't create a performance problem. A different implementation can be used to make sure that each thread is inspected only once.

3.4.2 Inthreads Program Analysis

Preserving sequential consistency during the compilation process is one of the major correctness concerns in parallel code compilation [27]. In our case, sequential consistency is provided by the combination of release-consistent execution and race-free (as defined by \textit{happens-before-1} relation) source program. To preserve sequential consistency, we have chosen a conservative approach that maintains the program order between the different communicating events that induce the \textit{happens-before-1} relation.

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 \( l_i \). 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.

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.

\textbf{Requirement 1.1} The compiler preserves the sequential execution semantics of each thread.
**Requirement 1.2** The compiler avoids introducing accesses to temporary variables in such a way as to make them shared.

**Requirement 1.3** The compiler preserves the order of synchronization instructions of each thread.

**Requirement 1.4** The compiler preserves the order of accesses to the shared variables with respect to synchronization operations.

**Requirement 1.5** The compiler avoids introducing new accesses to shared variables, except for those directly appearing in the program source.

**Requirement 1.6** The compiler avoids removing accesses to shared variables that are live at some synchronization statement.

**Requirement 2.1** For two concurrent statements $e_1$ and $e_2$, if variable $v_1$ is defined at $e_2$ 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 2.2** For 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 $e_2$.

Requirements 1.1 through 1.6 correspond to the optimization stage while requirements 2.1 and 2.2 correspond to the register allocation stage in the compilation process.

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.

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 $o_T$.

**Theorem 1:**
If the compiler obeys requirements 1.2, 1.3, 1.4, 1.5 and 2.1, 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 $a_1^{cp}$ and $a_2^{cp}$. There are two cases:
Case 1: $v_1$ and $v_2$ are the same variable $v$. If $v$ is not a temporary one, requirement 1.5 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 $v$ 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 1.3 and 2.2, a corresponding sequence consists in the compiled program between $a_1^{ce}$ and $a_2^{ce}$. Therefore, in any execution $a_1^{ce}$ and $a_2^{ce}$ would be ordered by happens-before-1, contradicting the assumption that there is a data race between them.

If $v$ is a temporary variable, then, because of requirement 1.2, 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 happens-before-1, and therefore, $v$ was shared when introduced, violating requirement 1.2.

Case 2: $v_1 \neq v_2$, but the compiler has mapped them to a single location $l$. According to requirement 2.1, 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 happens-before-1. 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 happens-before-1 and thus cannot form a data race.

Q.E.D.

Since the compiled program is data-race-free, its execution on RC 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 1.1, 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 1.6, 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 2.2, 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.
There is a different method proposed for analysis of explicitly-parallel programs: Delay Set Analysis by Shasha and Snir [37,25]. For a generic explicitly-parallel program, delay set analysis identifies a set of dependencies (delays) between program statements that must be enforced to preserve sequential consistency at the execution time. The set of delays is computed by finding critical cycles in the concurrent control flow graph extended with conflict edges. Figure 8 gives an example of a critical cycle and the corresponding delays.

Our execution model simplifies not only the hardware implementation [13], but also eliminates the need for the computationally-complex parallel code analysis algorithms, such as delay set analysis. For a data-race-free program with explicit synchronization, such as those targeted at Inthreads, data-independent shared variable accesses issued by a thread can be reordered as long as their order relative to corresponding synchronization statements is preserved. This is different from the general case of explicitly-parallel programs, where reordering accesses to shared variables may violate the interleaving semantics. Figure 8 gives an example of two shared variables $a$ and $b$ accessed by two parallel threads. In general case, the order between different accesses in each thread must be preserved to maintain sequential consistency. In a race-free program with explicit synchronization, happens-before-1 relation guarantees that either $d=a \rightarrow a=1$ and $c=b \rightarrow b=1$, or $b=1 \rightarrow c=b$ and $a=1 \rightarrow d=a$. This relation, dictated by explicit synchronization, remains true even if a pair of accesses in one (or both) threads is reordered.

### 3.4.3 Parallel Data Flow Analysis

Data flow analysis is a family of algorithms used to deduce data relations from the control flow structure of the program. Such analysis, for example, can be performed to find which assignments of a variable may reach a specified use of the variable or whether the value of the variable is a constant. Many optimization algorithms, such as dead code elimination, common sub-expression elimination and code motion, rely on the data flow information for their optimization process.

During the data flow analysis each statement in the program can either generate, kill or preserve a piece of data flow information. Variable definitions (write accesses) or variable uses (read accesses) are the most

<table>
<thead>
<tr>
<th>Thread #1 (initially a = b = 0)</th>
<th>Thread #2</th>
<th>— delay edge that must be preserved.</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>— conflict edge.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reordering one of the delays could lead to c=1 and d=0 - a result that violates sequential consistency.</td>
</tr>
</tbody>
</table>

**Figure 8 : Delays in a critical cycle**
trivial examples of such information. A typical implementation of the data flow analysis algorithm iteratively propagates the data-flow information through the CFG structure reaching the final solution [30]. Figure 9 demonstrates an example of a data flow problem of reaching definitions.

There are two main challenges of applying the traditional data flow to parallel code:

- A definition generated by a thread will not be considered available at the parallel threads since there is no sequential control flow path between the threads.
- A definition may be erroneously considered available at some execution point in a thread even if it was modified by a parallel thread.

Let us consider the first challenge. For the Inthreads execution model, if a value is written by one thread and is read by a parallel thread, there must be a happens-before relation between the events induced by the program order and pairs of matching synchronization operations. This relation also corresponds to a path along the CCFG edges: the program order corresponds to the sequential edges, while synchronization events correspond to specific synchronization edges.

To address such communication, we extend the data flow analysis to consider the set of synchronization edges. A definition of a shared variable \( v \) at execution point \( e_{\text{def}} \) will be marked as reaching the execution point \( e_{\text{use}} \) if there is a CCFG path connecting \( e_{\text{def}} \) and \( e_{\text{use}} \) and \( v \) is not redefined on that path. Such approach produced conservative results: all definitions that may reach point \( e_{\text{use}} \) will be reported, but some of the reported cases may never actually happen during the program execution.

Figure 10 gives an example of this approach. Propagating shared variable definition through synchronization edges will result in \( a_1 \) definition in thread #1 being marked as available at use point in
thread #3. Our analysis will mark both \( a_1 \) and \( a_2 \) as reaching the use point of \( a \).

The will now address the second challenge: a shared variable definition could be considered available at some execution point although in fact it was modified by a parallel thread. Such analysis is involved in code motion optimizations and usually refer to expressions rather than simple variable definitions. As an example, we could consider the question of whether \( a_1 \) equals \( b \) at line 4 (Figure 10), where traditional analysis could erroneously assume that this equality still holds.

In sequential case, a program statement \( \text{kills} \) (invalidates) a previously available expression if it modifies one of the expression operands. In a parallel program, a previously calculated expression may become unavailable when one of its operands is a shared variable, and the thread executed a synchronization operation with \( \text{release} \) semantics to give the other thread control over the shared variable.

To address such cases, we modified the compiler to invalidate references to shared variables at thread operations with \( \text{release} \) semantics, such as \( \text{INTH\_START()} \) and \( \text{INTH\_SET()} \). The following section gives examples of specific optimizations that use this approach to ensure transformation correctness.

For generic explicitly-parallel programs, there exist other parallel data flow analysis solutions [38,19]. Our extension to the calculation of reaching definitions is similar to [38]. Unlike the Parallel Fortran programs, considered by Grunwald and Srinivasan, our threads do not have implicit merging points such as "co-end" used to resolve conflicting accesses issued by parallel threads. There are several other differences between the execution models, such as the difference between the "copy-in / copy-out" semantics assumed by [38], and the release consistency model: whereas for release consistency a value may be modified after executing one of \( \text{INTH\_SET()} \) or \( \text{INTH\_START()} \) instructions, for copy-in/copy-out semantics the threads operate on local copies of the shared values and modifications are merged together at the end of the parallel block.

Knoop and Steffen [19] proposed methods to adopt sequential bit-vector data flow algorithms to parallel programs. They show that traditional data flow analysis techniques can be efficiently extended to parallel code by computing the properties over \( \text{interleaving predecessors (successors)} \) of a node in a CCFG graph. Their work doesn't consider the effects of explicit synchronization on the data flow analysis, but only refers to implicit synchronization performed at thread merging point.

In the following section we discuss some of the optimizations that rely on the data flow analysis and give examples on how the Inthreads compiler modifies them to preserve program semantics and optimization correctness.

3.5 Optimizations

Optimizations are code transformations performed by the compiler to increase the efficiency of the program. They target program performance, program size or both.

Optimization \( \text{safety} \) [30] is usually a precondition for a transformation. A compiler will aggressively apply an optimization, but never at the expense of making original code incorrect (with exception of special
compilation flags). This is also referred to as optimization conservativeness – meaning that for any given input, the transformed code will produce the same observable results.

While some optimizations, such as instruction scheduling or jump optimization, are generally flow-insensitive, others, such as dead code elimination, are categorized as flow-sensitive [30] since they require data flow information to solve the optimization problem. Such classification, usually used to determine the complexity of the problem, is also highly relevant to parallel code compilation. Whereas flow-insensitive optimizations bear little changes due to parallel code compilation, flow-sensitive optimization become inapplicable without adjustment.

In the following sub-sections we give examples of several of the commonly-used optimizations and describe the modifications we apply to adopt them to the parallel code compilation.

### 3.5.1 Dead Code Elimination

Dead code elimination is one of the basic optimization steps performed by a compiler. A program statement is considered to be dead if it produces a value never used in any of the following execution paths and has no side-effects (doesn't trigger an exception or change the control flow). The program statements that are not dead are called essential.

Dead code may originate from the program source as well as from the other code transformations that rely on this optimization step for post-processing.

The definition for dead code changes when considering parallel code: 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, naïvely applying sequential version of dead code elimination to parallel code may lead to incorrect results as some essential operations may be erroneously recognized as dead. Removing such accesses to shared variables would then violate requirement 1.6.

The dead code elimination algorithm requires data flow information such as reaching definitions and upward-exposed uses. The Inthreads compiler extends this algorithm with information obtained from parallel data flow analysis. Using parallel data flow analysis, the compiler marks shared variable definitions that may be accessed by subsequent uses in parallel threads as essential.

This modification guarantees correctness of this optimization when applied to parallel code: a statement will only be considered dead if it produces a value that is never used by any of the subsequently-executing statements including those belonging to parallel threads.

Consider the code example from Figure 4. Applying dead code elimination would start by marking instruction in line 7 as essential. Following the use-def chains for this instruction would result in marking instruction in line 5 as essential. At this stage, the definition in line 3 is marked as essential, but sequential data flow analysis would not recognize the definition of variable \(a\) in line 10 as being used, eventually marking this code as dead. Considering the explicit synchronization during data flow analysis, we mark the
definition in line 10 as being used by instruction in line 5, correcting the problem.

Dead code elimination can also be used on an CSSA representation of a parallel program [27]. In such case the algorithm uses use-def chains natively derived from the CSSA structure to recursively mark essential instructions.

3.5.2 Constant and Copy Propagation

For a given assignment \( x = y \), copy propagation optimization attempts to replace subsequent uses of \( x \) with \( y \). Constant propagation could be viewed as a special case of copy propagation where \( y \) is a constant.

In general case, this optimization can be formulated as a forward data flow problem [30]. Each statement in the CFG can generate a copy operation or kill other copy operations by modifying either of the participating variables. The set of copy operations available at a block entry point are defined to be the intersection of sets of copy operations available at the ends of the predecessor blocks.

For parallel code, copy propagation may repeat an access to a shared variable at a later point in the program order, potentially violating requirements 1.4 and 1.5. Since the optimization moves variable references in forward direction along the control flow structure, it has a potential to violate the \textit{shared access} \( \rightarrow \textit{release}() \) part of the \textit{happens-before} relation. Consider the code example in Figure 11. Processing this code sequentially, the optimizer could replace reference to variable \( a \) in \( c = a \) with \( b \) and reference to \( d \) in \( e = d \) with \( a \). Both would violate code correctness, assigning wrong value to \( c \) in line 4 and creating a data race between lines 5 and 9.

The Inthreads compiler restricts the optimization to maintain the original order between synchronization operations with \textit{release} semantics (INTH_SET() and INTH_START()) and the shared variable references. When the optimizer encounters such an operation, it conservatively invalidates copy operations related to the shared variables.

Such modification preserves correctness of copy propagation optimization for parallel code: a copy operation will be considered as available at an execution point only if neither of the participating variables is modified by any of the previously-executed statements including those belonging to a parallel thread.

We do not add any special handling of operations with \textit{acquire} semantics, such as INTH_WAIT(). Since the optimization moves variable references in forward direction, it has a potential to violate the \textit{shared access}

\[
\begin{array}{c}
\text{a = b} \\
\text{INTH_START()}
\end{array}
\]

\begin{tabular}{l|l}
\textbf{Thread #1} & \textbf{Thread #2} \\
(1) \ldots & (6) \textbf{a = \ldots} \\
(2) \ldots & (7) \textbf{d = a} \\
(3) \text{INTH_WAIT(1)} & (8) \text{INTH_SET(1)} \\
(4) \textbf{c = a} & (c = b) ? \\
(5) \textbf{a = \ldots} & (e = a) ? \\
\end{tabular}

\textit{Figure 11 : Copy propagation correctness}
For the example in Figure 11, the INTH_SET(1) operation in the second thread would invalidate the availability of \( d = a \) copy operation, and the INTH_START() would invalidate the \( a = b \) operation before proceeding to the following code.

An alternative solution is to use the parallel data flow analysis proposed in [19] where the problem is solved for generic explicitly-parallel programs by considering the *interleaving predecessors*. We've chosen to use our more conservative, but significantly simpler solution.

Lee, Midkiff and Padua [27] introduced a constant propagation algorithm for explicitly-parallel programs based on CSSA form, where they not only ensure correctness of the sequential propagation of constant values, but also allow intra-thread value propagation using def-use edges that connect statements of different threads. In [27] Lee also asserts that conventional copy propagation can be directly used without modifications if the parallel program is in CSSA form because CSSA form uses a local copy of a shared variable. Our approach can be used on traditional CFG representation without transforming it into an SSA form. It can also be applied at different stages of the compilation, including after register allocation.

### 3.5.3 Common Sub-Expression Elimination

*Common sub-expression elimination* (CSE) is one of the essential optimizations performed by sequential compilers. After calculating an expression, CSE attempts to replace subsequent recalculations of the same expression by a known result.

Similarly to copy propagation, naively applying CSE to parallel code results in a violation of the code correctness requirements 1.4 and 1.5. Figure 12 presents such an example. Replacing the calculation in \( D = B + C \) with a previously computed result \( A \) would be incorrect considering the parallel thread.

The Inthreads compiler solves this problem by using a solution similar to one applied to copy propagation: it invalidates expressions that access shared variables at synchronization points. Thus, the compiler will be able to substitute an expression by a previously-calculated value only if no one of the variables participating in expression is modified by any of the previously-executed statements including those belonging to parallel threads.

*Parallel data flow* [19] or *concurrent global value numbering* based on CSSA form [27] were also

<table>
<thead>
<tr>
<th>Thread #1</th>
<th>Thread #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( A = B + C )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>INTH_SET(1)</td>
<td>INTH_WAIT(1)</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( B = \ldots )</td>
</tr>
<tr>
<td>INTH_WAIT(2)</td>
<td>INTH_SET(2)</td>
</tr>
<tr>
<td>( D = B + C )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>

*Figure 12: Common sub-expression elimination*
suggested as a solution to parallel common sub-expression elimination. Those techniques, although more complex, provide solution for generic explicitly-parallel programs. For the purpose of the Inthreads compiler and the data-race-free program compilation, we've chosen a simpler approach described above.

3.5.4 Loop Invariant Code Motion

*Loop invariant code motion* optimization attempts to move computations that have the same value for each loop iteration to outside of the loop body. A statement can be identified as *loop invariant* if all arguments of the statement are constants, defined outside of the loop body or defined by a loop invariant statement inside the loop.

Obviously, above definition of *loop invariant* is incomplete when considering parallel programs. Although a shared variable is not modified inside a loop, it could be modified by a parallel thread. Moving a computation that depends on such a variable to outside the loop body would break code correctness and violate requirements 1.4, 1.5 and 1.6.

To solve this problem, the Inthreads compiler preserves the synchronization operations inside the loop. It then conservatively preserves the order between the synchronization operations and the shared variable accesses. Such modification to the optimization algorithm ensures that accesses to shared variables will not be erroneously recognized as *loop invariants*.

This problem can also be considered as a general code motion optimization when each computation is moved and combined at the earliest safe position. Knoop and Steffen consider both optimization correctness and optimization optimality [19,20] for the general code motion optimization. They use a complete parallel data flow analysis by considering all parallel predecessors (successors) of a basic block and calculating their effect on the data flow properties.
4 Register Allocation

Most modern compilers use an abstract internal representation of compiled code where values are expressed by \textit{virtual registers}. A \textit{virtual register} represents a scalar variable originating from the source code, or a temporary variable created during code transformations. Although virtual registers are an abstraction for the target machine registers, the number of virtual registers is usually assumed to be unlimited: together with source- and target-independent internal code representation this creates a convenient model that enables compilers to apply generic methods for code manipulation and optimization. Figure 13 gives an example of such a representation (a) and the corresponding machine code (b).

Register allocation is usually executed as one of the last compilation steps and is required to map possibly large number of used \textit{virtual registers} into machine-defined set of \textit{hardware registers}. The fixed number of hardware registers may be insufficient to accommodate all the variables used by the code, requiring compiler to introduce \textit{spill code} – additional code to store a value in memory after being defined and re-load it back before the subsequent usage.

Spilling has negative effect on the resulting code performance. Minimizing this overhead is one of the optimization problems of the register allocation phase. As with many other optimization problems, optimal register allocation was shown to be NP-complete [30].

Several different register allocation algorithms are known. Some of those algorithms apply different approaches to \textit{local} (within a single basic block or a loop) and \textit{global} (cross basic block) allocation. Other algorithms use a generic approach to solve the problem as a whole. As an example, current versions of GCC [47] use a local-global allocation algorithm.

The \textit{graph coloring} algorithm, introduced by Chaitin [9], is considered to be the dominant approach to register allocation. Recent years also brought new attempts to find algorithms for optimal or near-optimal solution of the allocation problem using Lagrangian [21] and Integer Linear Programming (ILP) [44] approaches.

In this work we investigate the aspects of register allocation for a light, multi-threaded architecture, where all threads concurrently access a common register file and may communicate using the machine registers. In the world of the parallel code compilation register allocation received little attention. Most SMT architectures provide a separate execution context for each thread – such environment does not require

```
... (set (vreg #100) (memref (offset (vreg #101) (const_int 64)))) (set (vreg #102) (add_op (vreg #100) (vreg #100))) (set (memref (offset (vreg #101) (const_int 68))) (vreg #102)) ... ld $4, 64($6) add $8, $4, $4 st $8, 68($6) ... a) internal representation
```

```
... b) assembler code
```

*Figure 13 : Example of an abstract representation with virtual registers*
modifications to the register allocation algorithms used for sequential code. It proved to be more appropriate in the world of embedded systems [45], where register allocation is required to handle threads with explicit context switching. We expect parallel execution to require generally more machine registers than corresponding sequential code. The quality of the register allocation for parallel threads is then a valid concern for us that will require an evaluation.

In the following sections we formally define the general case of register allocation for sequential code and extend our definitions to parallel code. Then, in Section 4.3 we discuss all the aspects of applying Chaitin's graph coloring allocator to the parallel code, including: building a parallel interference graph, coalescing, spill code generation and evaluation of the spill heuristic. In Section 4.4 we overview register allocation algorithms that aim at optimal or near-optimal register allocation and suggest methods for extending those algorithms to handle parallel code. Section 4.5 concludes the register allocation and summarizes the related work.

4.1 Register Allocation for Sequential Code

Register allocation in general is guided by the basic principle that a machine register can hold only a single value simultaneously. Therefore, the allocation must make sure that at each moment during the program execution distinct values will occupy distinct machine registers.

A compiler uses data flow analysis to determine the set of live variables that need to be allocated to registers at a specific program point. In Section 3.4.1 we gave a definition of the term live variable. In general, a variable is live at some location if there is a control flow path from that location to a subsequent variable use and the variable is not redefined on that path. This definition is equally applicable to virtual registers.

Intuitively, we can formulate sequential register allocation as follows:

*The compiler is required to assign machine registers to virtual registers in such a way that for each execution point no two live virtual registers get allocated with the same machine register.*

Such intuitive definition is too strict and does not account for possible correlation between different live variables. Consider two distinct values $a$ and $b$ that need to be stored to the register file at some point during the execution and to be used later. There must be a relative order between the definitions of $a$ and $b$: either $a$ was stored to the register file before $b$ or vice versa. At the moment when the later store operation occurs, the earlier value is live – it was stored to the register file to be (possibly) used later.

Register assignment criteria is based on the above observation. Let $E$ denote the set of all execution points (statements) in a program, and let $V$ denote the set of all virtual registers used in the program. We then define $Live(e) \subseteq V$ to be the set of live virtual registers for an execution point $e$ and $Defined(e) \subseteq V$ to be the set of virtual registers defined at $e$. For each virtual register $v$, register allocation assigns a hardware register or a
stack slot. We denote this assignment by predicate \( \text{reg}(v) \) (\( \text{reg}(v) \in R \cup S \), where \( R \) is the set of machine registers and \( S \) is the set of stack slots).

We can now use the above symbols to define sequential register allocation as follows:

**Definition 1: Sequential Register Allocation**

\[
\forall e \in E, \forall v_1 \in \text{Defined}(e), v_2 \in \text{Defined}(e) \cup \text{Live}(e): v_1 \neq v_2 \rightarrow \text{reg}(v_1) \neq \text{reg}(v_2)
\]  

Note that we also require different register assignment for different virtual registers defined at \( e \) – this is done for proper handling of instructions that define more than a single virtual register. For example, composite instructions may require temporary registers (scratch registers) during their execution.

### 4.2 Register Allocation for Concurrently-Executing Code

Let us now consider register allocation for parallel threads targeted at a shared context architecture, such as Inthreads: during the register allocation step, the compiler is required to allocate registers from a shared register file to a number of concurrently executing threads.

The same basic principle that distinct values must occupy different hardware registers is applicable to parallel code. Other basic concepts, such as the set of live variables at some point or the correctness of allocation and spilling, change considerably as compiler is required to consider the information on variables used by the parallel threads.

We will use the term **sequentially live** to denote virtual registers live at an execution point (the classical definition). We will also use the term **concurrently live** to denote virtual registers live at execution points that may execute concurrently with a given one. Intuitively, we can define register allocation for parallel code by extending the formulation from sequential compilation as follows:

"The compiler is required to assign machine registers to virtual registers in such a way that at each execution point no two sequentially or concurrently live virtual registers get allocated to the same machine register."

To model the parallel execution we will define a predicate \( \parallel \) as follows:

\( e_1 \parallel e_2 \) denotes that execution points \( e_1 \) and \( e_2 \) may be executed concurrently by the Inthreads machine. For the purpose of this discussion we could conservatively assume that any pair of execution points from two parallel threads satisfies the \( \parallel \) predicate. But our arguments are equally applicable to other, more accurate, definitions that could analyze synchronization statements inside a program to refine this set.

Formally, parallel register allocation can then be expressed as follows:

**Definition 2: Parallel Register Allocation**

Formulation in \( (1) \) holds, and

\[
\forall e_1 \parallel e_2 \in E, \forall v_1 \in \text{Defined}(e_1), v_2 \in (\text{Defined}(e_2) \cup \text{Live}(e_2)): v_1 \neq v_2 \rightarrow \text{reg}(v_1) \neq \text{reg}(v_2)
\]  

\( (2) \)
This formulation is also a symbolic representation of requirement 2.1 given in Section 3.4.2. It does not explicitly specify the fact that different threads may access a shared variable. In such cases the allocation must preserve this semantics by allocating the shared variable accesses in parallel threads with the same register or stack slot. We can add the following equation that conservatively defines the correctness requirement for a shared variable allocation:

\[
\forall e_1 \parallel e_2 \in E, \forall v_1 \in Defined(e_1), v_2 \in Live(e_2): v_1 = v_2 \rightarrow reg(v_1) = reg(v_2)
\]  

This formulation is a symbolic representation of the shared variable definition given in Section 3.4.1 and the requirement 2.2 given in Secion 3.4.2.

There are also other Inthreads-specific requirements given in Section 3.4.2 implying that register allocation must preserve release consistency and must not introduce data races on register file or memory accesses. These refer to code transformations that may be applied during register allocation (adding memory store and load instructions during spill code generation is one of such transformations).

Similarly to traditional register allocation, there are cases where number of machine registers is insufficient to hold all live virtual registers at some specific execution point. For such cases the register allocator must introduce spill code.

### 4.2.1 Register File Partitioning

One of the first questions in register allocation for parallel threads is how to partition the shared register file between the threads. We will discuss two possible strategies: explicit and automatic.

Explicit partitioning implies that definition of parallel blocks also provides the compiler with the definition of register file sections for each of the parallel threads. It is then compiler’s job to compile provided code so that each thread uses only the registers from the defined section of the register file. Such partitioning could provide the programmer or the parallelizing front-end with control over the intended resource partitioning and with performance tuning capabilities.

In the example in Figure 14 the first thread is to be allocated with registers 2-16 (integer registers) and the second thread is to be allocated with registers 17-31.

Although explicit partitioning seems like a straightforward solution for Inthreads, it has strong disadvantages:

- It requires user estimation of register pressure – estimation that could be inaccurate and that must be repeated after the code changes.
- It is machine-dependent.
- It requires special handling of read-only and shared variables used for communication (for example: it would be a waste to allocate different registers for the references to darr or N within different threads).
Automatic partitioning, on the other hand, implies that decision on allocating machine registers for parallel threads is solely carried out by the compiler. In this mode, the compiler is guided by the code structure itself: decisions on the amount of registers allocated to each thread and the use of read-only shared registers are performed automatically.

Due to the disadvantages of the static partitioning method, in this work we have chosen to explore the automatic partitioning solution based on graph coloring register allocator. Graph coloring as a general NP-complete problem has received a lot of attention in the past with numerous methods that offer approximated or accurate solutions [5]. In this work, we focus on the Chaitin-style graph coloring algorithm that is the dominant approach in register allocation. We discuss this solution in detail in the following sections.

4.3 Register Allocation by Graph Coloring

4.3.1 Background

Graph coloring paradigm was first applied to register allocation problem by G. J. Chaitin et al [9]. The idea is to represent the allocation problem as a K-color coloring problem of an interference graph - graph where nodes represent virtual registers and edges represent conflicts between them. A conflict between two virtual registers means that those registers can not be allocated with the same machine register (color).

Graph coloring problem in general is known to be NP-complete. Chaitin's register allocator uses a heuristic algorithm to solve this problem. To find a coloring, the algorithm uses the following basic principle:

**Figure 14: Example of register file partitioning**

<table>
<thead>
<tr>
<th>Explicit</th>
<th>Automatic</th>
</tr>
</thead>
</table>
| /* Thread #1 */
#pragma reg_range (2-16)
{
  int i;
  int maxx = 0;
  for (i = 0; i < N; i++) {
    maxx = max(maxx, darr[i].x);
  }
  ret->maxx = maxx;
}

/* Thread #2 */
#pragma reg_range (17-31)
{
  int j;
  int maxy = 0;
  for (j = 0; j < N; j++) {
    maxy = max(maxy, darr[j].y);
  }
  ret->maxy = maxy;
}
Given a graph $G$ and a node $n$, such that $n$ has less than $K$ neighbors, $G$ is colorable with $K$ colors iff $G\setminus\{n\}$ is colorable.

The algorithm consists of simplification steps: during each step an unconstrained (with less than $K$ neighbors) node is removed from the graph until there are no nodes left or all the nodes in the graph have at least $K$ neighbors (constrained). When only constrained nodes are left, the algorithm uses a heuristic step to remove one of them from the graph.

The algorithm flow (one of its versions) can be depicted as follows:

![Algorithm Flow Diagram]

- **Build**: During this step the interference graph is constructed.
- **Coalesce**: mov-related registers are coalesced to increase the possibility that they will be allocated to the same color.
- **Simplify**: Unconstrained nodes are removed from the interference graph. If only constrained nodes are left – one of them (according to spill heuristic) is marked as spill candidate and removed from the graph.
- **Select**: Nodes are restored back into the graph assigning each node a color that does not interfere with already colored neighbors. If a spill candidate can’t be allocated with a color – it is marked for spilling in Spill. The algorithm is finished when no spilling is required.
- **Spill**: Spill candidates that could not be colored are spilled. The algorithm is repeated again.

Most implementations of the above algorithm use live ranges (called webs in [30]) instead of virtual registers during the allocation. A live range is a subset of all definitions and uses of a virtual register, such that for each definition that belong to the range it also contains all the uses affected by the definition, and for each use that belongs to the range it also contains all the definitions that may affect the use. Working with live ranges, rather than with virtual registers, allows a compiler to eliminate false dependencies originating from the source code syntax and to decrease the constraints of the allocation problem.

A live range represents an independent part of a virtual register and can be seen as a variable by itself. Definitions that we used for a live variable or a shared variable, are also applicable to a live range. Unless use of live range term is explicitly required by the context, in the rest of this work we use the terms variable and live range interchangeably.

Figure 16 illustrates an example of live ranges in a program. A single virtual register $a$ is split into two live ranges ($a_1$ and $a_2$). Each live range is then treated independently during register allocation run and can be
When building an interference graph for sequential code, two live ranges $a$ and $b$ are marked as interfering if $a$ is live during the definition of $b$ (or vice versa). A more exact definition also requires that $b$ is not a copy of $a$ at the point of its definition (that is $b$ is not defined by “mov b, a” instruction) – such specialization distinguishes the cases where live ranges having the same value are not considered to be interfering. In fact, it is also required for coalescing, discussed later.

4.3.2 Parallel Interference Graph

The first step of the graph coloring algorithm is to translate intermediate code representation into an interference graph. Conceptually, interference represents a condition which we symbolically formulated in (1) and (2): two live ranges are said to interfere if they cannot be allocated with the same machine register.

We will use the terms sequential interferences to denote the interferences as specified by formulation (1), and parallel interferences to denote parallel interferences added by formulation (2).

The graph building algorithm consists of two stages: building the live ranges (nodes), and calculating the interferences between the live ranges (edges). Both of those stages need to be modified to comply with the parallel register allocation requirements defined in Section 4.2.

First, we consider the task of building the live ranges for parallel code.

Live range building is essentially the same data flow algorithm as used in reaching definitions data flow problem [30]: a virtual register definition and a use will belong to the same live range if there is a control flow path between the definition and the use and the virtual register is not reassigned on that path. In a parallel program though, the compiler needs to consider not only the possible sequential execution paths, but concurrently executing ones as well. We address this problem by considering synchronization edges during the data flow analysis (Section 3.4.3). If a value assigned to a shared variable can be accessed by a parallel
thread, both the variable definition and the subsequent variable use will form a single live range. Such construction ensures that both threads will later share the allocated hardware register or a stack slot and it corresponds to the formulation given in (3).

In the example in Figure 17 the definition of the shared variable \( a \) will be connected through the synchronization edges with the use of the variable in the parallel thread to form a single live range.

The next step of building a parallel interference graph is to collect the interferences between the live ranges. We modified the graph building algorithm to extend the set of computed sequential interferences with parallel interferences.

Figure 18 presents pseudo code of the modified algorithm. The traditional implementation of this

```plaintext
procedure CollectRegConflicts (cfg)
Conflicts = CollectSequentialConflicts(cfg);
foreach \(<E_1,E_2>\) in GetConcurrentStatements()
    foreach reg_range_1 in Defined(E_1) do
        foreach reg_range_2 in (Live(E_2) \cup Defined(E_2)) do
            if (reg_range_1 != reg_range_2) do
                Conflicts U= <reg_range_1,reg_range_2>
            done
        done
    done
    foreach reg_range_1 in Defined(E_1) do
        foreach reg_range_2 in Live(E_1) do
            if (reg_range_1 != reg_range_2) do
                Conflicts U= <reg_range_1,reg_range_2>
            done
        done
    done
return Conflicts
end
```

Figure 18 : Algorithm to collect intra-thread live range conflicts
algorithm is denoted by \( \text{CollectSequentialConflicts}() \) [30]. We use \( \text{GetConcurrentStatements}() \) procedure to implement the \(||\) operator described earlier. The procedure returns sets of concurrent statements. In our evaluation we used a conservative implementation of this procedure that marks all statements of a thread to be concurrent with all statements of a parallel thread.

Despite the conservatism, this approach produces accurate results for most of the fine-grained parallel cases we encountered as the threads would usually execute majority of the code concurrently and use ordered execution only for short periods. Figure 19 presents typical examples of such cases: Figure 19a lists parallel data-independent code segments that synchronize only at termination, Figure 19b lists parallel loop iterations that synchronize only when acquiring the next loop index. Figure 19c presents a synthetic example with synchronization-induced order between parallel blocks where our approach could produce over-pessimistic results. For such cases, our CCFG building algorithm can be augmented with a synchronization analysis algorithm such as the one described in [33].

Figure 20 gives a code example with the corresponding interference graph. Variable \( a \) is shared by both threads, while the rest of the variables are accessed only by one of the threads. Solid lines represent sequential interference computed by traditional interference graph building algorithm while dashed lines represent

<table>
<thead>
<tr>
<th>a) data-parallel computation</th>
<th>b) parallel loop</th>
<th>c) synchronized computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTH_START(1, start_2);</td>
<td>INTH_START(1, start_2);</td>
<td>INTH_START(1, start_2);</td>
</tr>
<tr>
<td>#pragma inthread</td>
<td>#pragma inthread</td>
<td>#pragma inthread</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>for (pin = 0; pin &lt; N; pin++)</td>
<td>while (1) {</td>
<td>for (i=1; i &lt; N; i=2) {</td>
</tr>
<tr>
<td>x = blocks[pins[pin]].x;</td>
<td>INTH_WAIT(1);</td>
<td>a[i] = VALUE_A(a[i-1], b[i]);</td>
</tr>
<tr>
<td>if (x &lt; xmin) {</td>
<td>mylink = link;</td>
<td>INTH_SET(1);</td>
</tr>
<tr>
<td>xmin = x;</td>
<td>if (link)</td>
<td>INTH_WAIT(2);</td>
</tr>
<tr>
<td>} else if (x &gt; xmax) {</td>
<td>INTH_SET(1);</td>
<td>b[i] = VALUE_B(b[i-1], a[i]);</td>
</tr>
<tr>
<td>xmax = x;</td>
<td>if (!mylink) break;</td>
<td>INTH_SET(3);</td>
</tr>
<tr>
<td>}</td>
<td>v1 += VALUE_OF(mylink);</td>
<td>INTH_WAIT(4);</td>
</tr>
<tr>
<td>bbox-&gt;xmin = xmin;</td>
<td>INTH_WAIT(2);</td>
<td>INTH_SET(5);</td>
</tr>
<tr>
<td>bbox-&gt;xmax = xmax;</td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>INTH_WAIT(1);</td>
<td>return v1 + v2;</td>
<td>return;</td>
</tr>
<tr>
<td>return bbox;</td>
<td>start_2:</td>
<td>start_2:</td>
</tr>
<tr>
<td>#pragma inthread</td>
<td>#pragma inthread</td>
<td>#pragma inthread</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>for (pin = 0; pin &lt; N; pin++)</td>
<td>while (1) {</td>
<td>for (i=2; i &lt; N; i=2) {</td>
</tr>
<tr>
<td>y = blocks[pins[pin]].y;</td>
<td>INTH_WAIT(1);</td>
<td>a[i] = VALUE_A(a[i-1], b[i]);</td>
</tr>
<tr>
<td>if (y &lt; ymin) {</td>
<td>mylink = link;</td>
<td>INTH_WAIT(3);</td>
</tr>
<tr>
<td>ymin = y;</td>
<td>if (link)</td>
<td>INTH_SET(4);</td>
</tr>
<tr>
<td>} else if (y &gt; ymax) {</td>
<td>INTH_SET(1);</td>
<td>b[i] = VALUE_B(b[i-1], a[i]);</td>
</tr>
<tr>
<td>ymax = y;</td>
<td>if (!mylink) break;</td>
<td>INTH_SET(3);</td>
</tr>
<tr>
<td>}</td>
<td>v2 += VALUE_OF(mylink);</td>
<td>INTH_WAIT(4);</td>
</tr>
<tr>
<td>bbox-&gt;xmin = xmin;</td>
<td>INTH_WAIT(2);</td>
<td>INTH_SET(5);</td>
</tr>
<tr>
<td>bbox-&gt;xmax = xmax;</td>
<td>}</td>
<td>INTH_WAIT(5);</td>
</tr>
<tr>
<td>INTH_SET(1);</td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>INTH_HALT();</td>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Figure 19: Concurrent execution examples
parallel interference collected by the algorithm presented in Figure 18.

4.3.3 Coalescing

Coalescing is a step in the register allocation algorithm when non-interfering live ranges are merged together. Such transformation is justified when there are mov operations that copy one live range into another – if both are allocated with the same machine register, the mov instructions can be eliminated. Two non-interfering live ranges are usually chosen for coalescing using a heuristic that eliminates maximal number of mov instructions while preferably preserving graph colorability (that is, merging two nodes in a graph should not introduce a constrained node that would later require spilling).

One could look at coalescing as a form of copy propagation, as such it will require additional considerations to preserve compilation correctness. Let us consider the example in Figure 21a. The assignment local = a is a subject to coalescing. Traditional coalescing could, for example, generate the register allocation presented in Figure 21c.

Coalescing the two variables actually moved the read access of the shared variable a (now allocated to reg1) below the INTH_SET() instruction. Such code motion violates the original read(a)→release(1) ordering, potentially causing a data race if a is modified by a parallel thread.

However, with our method of construction of the parallel interference graph this violation will not occur. Let us formally prove this.

Theorem 2:

Applying coalescing to the parallel interference graph constructed by the algorithm in Section 4.3.2 will not introduce a data race on shared variable accesses.

Proof:

/* a - shared var. */
local = local var. */
INTH_WAIT(1);           // Thread #1
a = a + 1;
local = a;
INTH_SET(1);
smlh1 = local + 10;

/* reg1 - shared reg. */
reg2 - local reg. */
INTH_WAIT(1);           // Thread #2
reg1 = reg1 + 1;
reg2 = reg1;
INTH_SET(1);
reg3 = reg2 + 10;

Thread #1
Thread #2
a = ... d = ... 
INTH_SET(1) INTH_WAIT(1)
b = ... e = d + a 
c = ... a = a + 1 
INTH_WAIT(2) INTH_SET(2)
c += b + a ... = e

Figure 20: Interference graph example

Figure 21: Register coalescing in parallel code
Assume that there exist a `mov a, b` instruction, such that coalescing `a` and `b` introduces a data race. Namely, there exist concurrent execution points `e_1` and `e_2`, such that `e_1` is an access to `a` and `e_2` is an access to `b`, and at least one of them is a `write` access.

Since `e_1 \parallel e_2`, our parallel interference graph building algorithm marks all live ranges defined at `e_1` as interfering with those live or defined at `e_2` (and vice versa). Therefore the parallel interference graph will contain an interference edge between `a` and `b`.

Based on the above, we conclude that the `mov` instruction could not been considered for coalescing since coalescing is applied only to a pair of non-interfering live ranges.

Q.E.D.

4.3.4 Spill Code Generation

In some cases, the number of machine registers is not sufficient to accommodate all the virtual registers in the program. To handle the problem, the compiler generates spill code – additional `store` and `load` instructions that temporary place values on the stack to be reloaded into a machine register when necessary.

Chaitin et al [9] originally proposed a spilling method called `spill-everywhere`. In this method, a spill candidate is spilled immediately when encountered. For a spilled live range, stores are inserted after each definition and loads are placed before each use. Later proposals introduced improvements to this basic behavior, such as optimistic coloring [7] where spilling decision is delayed till the `Select` phase, and interference region spilling [4] that attempts to reduce the amount of spill code generated by the allocator.

For parallel code, spilling shared variables requires additional considerations to preserve correctness. Spilling must support requirements 1.2 through 2.2, ensuring that communication through shared variables proceeds correctly and avoiding introducing race conditions on either memory or register file accesses.

To preserve communication through shared variables, spilling must be performed consistently in all the threads: a shared variable spilled in a thread must be properly reloaded when referenced in one of the parallel threads.

Race conditions may result from improper placement of new memory access instructions when spilling variables accessed by parallel threads: Section 4.3.4.2 gives examples of such cases.

In the following sections we consider two possible spill methods: `spill-everywhere` and interference region spilling. For each of the methods we review the relevant implications of the parallel code compilation. Then, in section 4.3.4.4, we evaluate the `spill cost heuristic` that drives the spilling decisions of the graph coloring allocator.

4.3.4.1 Spill Everywhere

The `spill-everywhere` method processes each basic block independently. For each variable marked for spilling, the allocator inserts a store after each definition and a load before each use. This basic approach is
applied with a simple optimization that says that there is no need to add a load between close references to the spilled variable. Subsequent references are said to be close, if no other variable dies between them (a variable dies at a use after which it is no longer live [30]).

Due to its locality, this approach is also valid for parallel code.

Theorem 3:

Spill-everywhere method preserves parallel code correctness: it conforms to parallel register allocation formulation specified in Section 4.2 and does not introduce data races.

Proof:

Our proof is based on the CCFG structure (Section 3.3.1). Recall that CCFG construction starts a new basic block whenever it encounters an instruction with acquire semantics, such as INTH_WAIT(), and terminates the basic block after an instruction with release semantics, such as INTH_SET() and INTH_START().

For each spilled variable, spill-everywhere allocates a memory location \( s \). It also generates smaller live ranges, each being live only at a single basic block that accesses the original variable. The shorter live ranges are treated independently during the subsequent allocation passes.

A variable assigned by a thread will be stored to the memory location \( s \) just after the variable definition and before any INTH_SET() instruction in that basic block. A read access will be preceded by a load of the variable from the memory \( s \); this load is placed just before the first read access in that basic block, and thus will be placed after any INTH_WAIT() instruction in that block.

Such implementation makes sure that there are no new or deleted accesses to shared variables, it preserves the order of variable access relative to synchronization operations and the original happened-before-1 ordering of the basic blocks. Thus it satisfies the correctness requirements 1.1 through 1.6 specified in Section 3.4.2. Together with subsequent rebuild of the interference graph it also satisfies the requirements 2.1 and 2.2 also formulated by (2) and (3).

Q.E.D.

<table>
<thead>
<tr>
<th>Before allocation</th>
<th>After allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread #1</strong></td>
<td><strong>Thread #2</strong></td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( b = b + 1 )</td>
<td>( \text{addi} )</td>
</tr>
<tr>
<td>( \text{basic block boundaries} )</td>
<td>( 2, \text{int}h\text{.wait} 2 )</td>
</tr>
<tr>
<td>( \text{INTH_WAIT}(1) )</td>
<td>( \text{int}h\text{.wait} 1 )</td>
</tr>
<tr>
<td>( a = b )</td>
<td>( \text{st} \text{ } 2, 64(\text{sp}) )</td>
</tr>
<tr>
<td>( \text{INTH_SET}(2) )</td>
<td>( \text{int}h\text{.set} 2 )</td>
</tr>
<tr>
<td>( \text{basic block boundaries} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( c = c + 1 )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>

| **Thread #1**     | **Thread #2**   |
| \( \ldots \)      | \( \ldots \)    |
| \( \text{addi} \)      | \( \text{int}h\text{.wait} 2 \) |
| \( 2, \text{int}h\text{.set} 1 \) | \( \text{ld} \text{ } 4, 64(\text{sp}) \) |
| \( \text{st} \text{ } 2, 64(\text{sp}) \) | \( \text{int}h\text{.set} 1 \) |
| \( \text{int}h\text{.set} 2 \) | \( \text{addi} \text{ } 4, 4, 1 \) |

Figure 22: Example of spill code generation

36
Figure 22 gives an example of a shared variable being spilled. Note that each thread uses a different hardware register to access the shared variable ($2 and $4) and the actual sharing is performed through the memory rather than a machine register - such allocation is a result of the cross-interference between the split live ranges discussed above. The memory load and store instructions are placed after and before the corresponding synchronization instructions.

Figure 23 gives an example of a spilled read-only variable used by parallel threads. The references to the variable in different threads were allocated to different registers ($2 and $4 respectively). The memory location, holding the value of the variable, is concurrently read by both threads.

Although spill-everywhere offers a convenient and relatively simple method for generating spill code for both sequential and parallel cases, it has an important drawback: it generates spill code in a uniform manner, possibly adding spill code in places where none was actually required. In the following section we will consider interference region spilling that aims at eliminating this behavior.

4.3.4.2 Interference Region Spilling

This method attempts to reduce the amount of emitted spill code by spilling only a subset of the references to a variable. Instead of spilling the whole range, it chooses a color that has minimal interference with the spilled live range and adds spill code only in places where the selected color is already in use by some other live live range. Additional reload code is added at interference boundaries [4] - points where the chosen color becomes available while the spilled live range is live. To make sure that reload code always accesses the most recent value of the variable, stores to the memory are placed at all variable definitions that may reach any of the reload statements (note that these stores would always be added by the spill-everywhere method).

This method is applied only when spilling at interference boundaries is actually cheaper, according to the estimated overhead of the additional reload statements, than spilling the whole live range.

Figure 24a presents a sequential code example and Figure 24b depicts the corresponding interference graph (taken from [4]). The allocator is required to spill A to achieve 2-color allocation. Figure 24c gives the code listing after inserting the spill code, where instructions in lines 13 and 15 are only added when applying spill-everywhere method and are eliminated when applying interference region spilling. This result is...
achieved by spilling the $A \leftrightarrow C$ interference.

Interference region spilling is heavily based on the data flow information and, thus, is profoundly affected by parallel code compilation. We will address three major aspects of this approach affected by parallel code compilation: a) estimation of profitability based on variable liveness; b) detecting interference boundaries in parallel code; c) actual spill code generation and correctness of the memory access instruction placement.

The algorithm uses the information on the set of live variables at each program statement to estimate profitability of the interference region spilling before the actual spilling and to calculate the set of already allocated colors during the spill code insertion. To accurately represent the variable liveness information in a parallel program, we must extend the algorithm to consider not only the sequentially live variables, but also the concurrently live ones.

For two sequentially live variables, the interference boundary is defined to be the location in code where one of them dies while the other remains live. Such condition can be locally detected by inspecting variable uses at a specific program statement and liveness information after the statement. In parallel case, death of a concurrently live variable is propagated to a parallel thread and is detected at a block boundary with a synchronization statement.

Let us now examine correctness of the actual spill code generation.

In case of a private variable used only by a single thread, the interference region spilling behaves similarly to sequential case. Applying this method to a shared variable is more challenging since interference region spilling may introduce a reload of the shared variable in a location where no access to the variable existed previously, potentially creating a race condition. Figure 25 gives a simple example of a shared variable $a$ being spilled inside a thread. Note the reload access in line 7 that could be added by the interference region spilling when the selected hardware register used by variable $b$ becomes available (line 7 is an interference
Adding access to \( a \) at line 7 can introduce a race condition if \( a \) is concurrently modified by another thread.

For each variable, the interference region spilling algorithm makes a cost-dependent decision on whether to use spill-everywhere or region spilling. Such flexibility suggests the simplest solution for the problem in Figure 25 - using spill-everywhere method is safe for spilling a shared variable (we will name this method selective interference region spilling).

This solution can be further refined by applying spill-everywhere only at basic blocks that belong to parallel code. We could say that interference region boundary in such case is placed at the first sequential basic block after the thread. A reload must be placed at that block if the shared variable is live and the selected color is available, making sure that shared variable uses inside the thread and the uses in the following sequential code will form separate live ranges with separate spilling methodologies.

Figure 26 gives an example of applying the refined solution to a spilled shared variable \( a \). The allocator used spill-everywhere method for the shared variable references inside the threads, and interference region spilling for the following sequential code.

<table>
<thead>
<tr>
<th>Thread #1</th>
<th>Thread #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) ( a = \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>(2) <code>&lt;store a&gt;</code></td>
<td>( \ldots )</td>
</tr>
<tr>
<td>(3) <code>INTH_SET(1)</code></td>
<td><code>INTH_WAIT(1)</code></td>
</tr>
<tr>
<td>(4) ( b = \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>(5) ( \ldots = b )</td>
<td>A ( a = \ldots )</td>
</tr>
<tr>
<td>(6) <code>&lt;reload a&gt;</code></td>
<td><code>&lt;store a&gt;</code></td>
</tr>
<tr>
<td>(7) <code>INTH_WAIT(2)</code></td>
<td><code>INTH_SET(2)</code></td>
</tr>
<tr>
<td>(8) ( c = a + 1 )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>

**Figure 25:** Race condition due to shared variable reload

<table>
<thead>
<tr>
<th>Thread #1</th>
<th>Thread #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_1 = \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td><code>&lt;store a&gt;</code></td>
<td><code>INTH_WAIT(1)</code></td>
</tr>
<tr>
<td><code>INTH_SET(1)</code></td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( b = \ldots )</td>
<td>( a_2 = \ldots )</td>
</tr>
<tr>
<td><code>INTH_WAIT(2)</code></td>
<td><code>&lt;store a&gt;</code></td>
</tr>
<tr>
<td><code>&lt;load a&gt;</code></td>
<td><code>INTH_SET(2)</code></td>
</tr>
<tr>
<td><code>c = a_1 + 1</code></td>
<td><code>INTH_SET(4)</code></td>
</tr>
<tr>
<td><code>INTH_WAIT(4)</code></td>
<td><code>INTH_HALT()</code></td>
</tr>
</tbody>
</table>

**Figure 26:** Selective interference region spilling
As an alternative approach, we could use data flow analysis when adding reloads: given an interference boundary \( b \) and a set of spilled variable uses \( E \), we will place a reload before a use \( u \in E \) if there is a path from \( b \) to \( u \) such that \( u \) is the first use on the path and the variable is not assigned on the path. This approach is similar to \textit{spill-everywhere}: it adds memory accesses only at the same blocks where the shared variable was already accessed before. In many cases it will result in the same load instruction placement as the \textit{spill-everywhere}, but for some cases it could produce less conservative results.

Let us now consider spilling a read-only shared variable. Although none of the threads modifies the variable, adding reloads may actually introduce a data race. Figure 27 presents an example of such condition. After the spilling, both threads define and use the variable \( a \). Since the threads are connected by synchronization, parallel data flow detects both definitions of \( a \) in lines 4 and 10 as reaching the use in line 7, subsequently forming a single live range. In such case, the reload added in line 4 and the load in line 10 generate a race condition on register file access.

Such race condition is forbidden in the Inthreads architecture, regardless of the fact that both statements write the same value into the machine register holding \( a \).

Note that the race occurs only because both statements form a single live range. Such pseudo-communication is a result of introduction of the spill code and the following data flow analysis that spuriously assumes that the variable is intended for thread communication.

The allocator can effectively avoid such changes in code semantics by renaming the new live ranges created by loading a spilled variable. In our example, renaming \( a \) in lines 10-11 to \( a_1 \), would eliminate the symbolic relation between the variable definition in line 10 and the variable used in line 7. Using different live ranges during the following allocation passes will then eliminate the data race.

4.3.4.3 Short Live Ranges

A notable peculiarity of the register allocation for parallel code can be observed in the case of short live ranges. A short live range is one where all references are close so that no other live range dies between the range definitions and uses.

A short range has special meaning during spilling: since spilling a short live range does not free any
machine registers, its spill cost is defined to be infinite. In practice, it would be impossible to have sequential code that produces an interference graph where all the nodes are constrained and all of them represent short live ranges. However, this situation can happen in parallel code compilation. Figure 28 presents a graphical example of such condition. Each thread has a set of short live ranges. Due to parallel execution, the sets interfere with each other, making each short live range constrained due to the high number of interferences.

Although a parallel interference graph is always colorable with \( \frac{x}{x_1} k \) colors (where \( x \) is the maximal number of registers used by an instruction and \( k \) is the number of parallel threads), greedy color selection algorithm used by graph coloring register allocation may fail to find that coloring. Consider greedy color selection for the graph presented in Figure 29: if the order of color assignment is \( v_0, u_0, v_1, u_1 \ldots \) the algorithm would require \( n \) colors (where \( n \) is the number of nodes in the graph).

We modified the graph coloring algorithm so that when it encounters an interference graph where all nodes are constrained short live ranges, it removes a node from the graph and enqueues it for color assignment in Select phase.

To ensure coloring for any parallel interference graph we can control the order in which short register ranges are removed from the graph so that we color the live ranges of one thread before coloring the live ranges of a parallel thread.

4.3.4.4 Evaluating Spill Cost Heuristic

Behavior of the graph coloring allocator is governed by spill cost heuristic \( h(v) \). For a graph where all nodes are constrained, the allocator chooses a spill candidate by finding a live range with minimal \( h(v) \).

A lot of effort was invested in identifying good spill heuristic [6,7]. In general, it was shown that no single formula dominates all others in all cases. In their work Bernstein et al have proposed using “best of three” method – choosing best of three proposed cost functions in order to span wider range of good decisions and minimize the NP-noise [7] – large variations in spill costs that arise when even slightly modifying the
algorithm.

The spill cost of a live range $v$ is usually calculated by a formula of the kind:

$$h(v) = \frac{\text{cost}(v)}{\text{degree}(v)},$$

where $\text{degree}(v)$ denotes the number of neighbors for $v$ in the graph, and $\text{cost}(v)$ is calculated as:

$$\text{cost}(v) = \sum_{i \in \text{Instructions}} 10^{\text{depth}(i)}.$$

The expression $10^{\text{depth}(i)}$ represents a static estimation of the number of times instruction $i$ is expected to execute where $\text{depth}(i)$ denotes the depth of the loop nesting.

Although an interference graph can theoretically have an arbitrary structure, the interference graphs of sequential programs are usually sparse. For architectures with large regular register files, such as MIPS and Alpha, the number of neighbors for each node in the interference graph rarely exceeds the number of available registers [7]. The situation is different in case of parallel code: number of nodes having high degree is considerably higher, increasing the number of heuristic-driven decisions required for choosing a spill candidate.

In a parallel interference graph the degree of a node comprises two counters: conflicts between sequentially live ranges ($S\text{deg}$) and conflicts between concurrently live ranges ($C\text{deg}$). The spill cost heuristic function can then be expressed by:

$$h(v) = \frac{\text{cost}(v)}{(S\text{deg}(v) + C\text{deg}(v))}$$

To determine the influence of the concurrently live conflict count on the quality of the heuristic, we introduce relative weights in the above formula as follows:

$$h(v) = \frac{\text{cost}(v)}{(S\text{deg}(v) + f \times C\text{deg}(v))}$$

Total spill cost of the allocation is defined to be the sum of costs for newly introduced spill code instructions (where load is assumed to be twice the weight of a store), minus the cost of mov instructions removed by coalescing. In order to estimate a single instruction cost, we use statically-computed block execution frequencies (profile-driven execution frequencies can be used as well).

We used a set of allocation problems from SPEC2000 integer benchmarks [46] and from Mediabench suite [49]. The procedures were chosen by analyzing the benchmark profiles and then manually parallelized. In most of the parallelized procedures, parallel code segments execute with little synchronized serialization, so that our conservative implementation for the $\|$ operator in fact produces accurate results. In our measurements we use spill-everywhere method since interference region spilling is not well suited for such evaluations. With interference region spilling, the spill cost of a live range depends on the previously allocated colors. Spill cost heuristic is used at the stages prior to color assignment and can not fully reflect such behavior.

Due to the heuristic nature of the algorithm, the spill cost achieved by the register allocation may be

42
search the space of numeric functions operating on given parameters, such as spill cost, number of sequential interferences and number of parallel interferences. We used Genetic Programming (GP), which offers a way to evolve specific object properties by means of probabilistic crossover and mutation.

Similarly to [6], the observed effect of different f values is rarely consistent: no single value dominates the others. Our initial estimation was that the algorithm will perform better when ignoring the number of parallel interferences (with f=0) because spilling a node inside a thread does not reduce this number: loading a spilled variable still requires a temporary register. The results in Table 1 do not support this estimation. This can be explained by the fact that spilling inside a thread indirectly reduces the register pressure at parallel threads by reducing the total number of machine registers required by the thread using the spilled variable. In fact, we see that f=0.5 often provides good results.

A more general approach to the problem of finding an optimal heuristic for spill code generation is to search the space of numeric functions operating on given parameters, such as spill cost, number of sequential interferences and number of parallel interferences. We used Genetic Programming (GP), which offers a suitable model for exploring very large solution space [40]. GP uses a set of object instances (population) to evolve specific object properties by means of probabilistic crossover and mutation.

We used a population of 400 arithmetic expression trees over integers and known parameters. During each generation, new expressions are generated by depth-fair crossover [17] between two expression trees, and 5% of the expressions undergo random mutations.

Table 1: Allocation cost as function of register pressure

<table>
<thead>
<tr>
<th>REGS</th>
<th>Allocation cost</th>
<th>Values</th>
<th>TFValues</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>52.50</td>
<td>3360</td>
<td>3455</td>
</tr>
<tr>
<td>0.5</td>
<td>52.00</td>
<td>3360</td>
<td>3455</td>
</tr>
<tr>
<td>1</td>
<td>52.50</td>
<td>3360</td>
<td>3455</td>
</tr>
<tr>
<td>1.5</td>
<td>52.50</td>
<td>3360</td>
<td>3455</td>
</tr>
<tr>
<td>2</td>
<td>52.50</td>
<td>3360</td>
<td>3455</td>
</tr>
</tbody>
</table>

Table 1 lists the results for each benchmark with varying f and varying number of available registers. Similarly to [6], the observed effect of different f values is rarely consistent: no single value dominates the others. Our initial estimation was that the algorithm will perform better when ignoring the number of parallel interferences (with f=0) because spilling a node inside a thread does not reduce this number: loading a spilled variable still requires a temporary register. The results in Table 1 do not support this estimation. This can be explained by the fact that spilling inside a thread indirectly reduces the register pressure at parallel threads by reducing the total number of machine registers required by the thread using the spilled variable. In fact, we see that f=0.5 often provides good results.

A more general approach to the problem of finding an optimal heuristic for spill code generation is to search the space of numeric functions operating on given parameters, such as spill cost, number of sequential interferences and number of parallel interferences. We used Genetic Programming (GP), which offers a suitable model for exploring very large solution space [40]. GP uses a set of object instances (population) to evolve specific object properties by means of probabilistic crossover and mutation.

We used a population of 400 arithmetic expression trees over integers and known parameters. During each generation, new expressions are generated by depth-fair crossover [17] between two expression trees, and 5% of the expressions undergo random mutations.
The expression parameters include: $c$ – estimated spill cost of the live range, $r$ – number of sequential interferences, $p$ – number of parallel interferences, $a$ – number of coalesced live ranges, $d$ – number of deaths spanned by the live range. For an example, the default spill cost heuristic presented above is expressed as $(c/(r+p))$. We used total spill cost of the training set as a measure of fitness of an expression.

Figure 30 shows the spill cost of the best expression found by this algorithm over 50 generations for three independent runs (the number of available machine registers was set to 16). Assuming there exists an “optimal spill heuristic for the graph coloring algorithm for parallel code”, Figure 30 shows that the default spill heuristic could be more than 10% worse than the optimum.

On the other hand, the default heuristic offers reasonable allocation quality with very short expressions, as opposed to sometimes very long expressions produced by the genetic search. Figure 31 shows the relation between the depth of the expression tree and the achieved allocation cost.

Some examples of the best expressions found by the genetic algorithm are presented in Table 2. Note that the same pattern $(c/d)*p$ repeats itself in different forms. The above form has some clear principles behind it. $c$ (estimated spill cost of a live range) is the base of the expression. It is divided by $d$ that represent the size of the live range: live ranges that span larger code segments usually also span more deaths of other, shorter live ranges. The multiplication by $p$ (number of parallel interferences) represents a “fairness” factor in the order in which nodes are removed from the graph. Imagine an interference graph constructed for two symmetrical

<table>
<thead>
<tr>
<th>Expression</th>
<th>Cost</th>
<th>Expression length</th>
<th>Expression length limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(((c / d) * p) + c)$</td>
<td>767025</td>
<td>23</td>
<td>≤ 30</td>
</tr>
<tr>
<td>$(((c / d) - ((255 + d) / ((81 / 81) / d))) * p) + c)$</td>
<td>766772</td>
<td>56</td>
<td>≤ 60</td>
</tr>
<tr>
<td>$(((243 * c) - ((98 * p) * 154)) + ((p + (((244 * c) / d) + ((c * p) + 63)) / d)) * 235))$</td>
<td>765631</td>
<td>92</td>
<td>≤ 100</td>
</tr>
</tbody>
</table>

Table 2: Best expressions of specified length
threads A and B: removing a node that belongs to thread A will reduce the number of parallel interferences for all the nodes that belong to thread B. In the next step, the allocator will then favor the nodes from B over those from A.

To validate the GP algorithm results, we ran register allocation with the best expression of length 100 on code examples from an evaluation set (different from the training set). Although the total spill cost of the evaluation set was by 6% in favor of the GP expression, it did not absolutely dominate the default behavior and in some cases produced worse results (Figure 32).

To further estimate the effect of the parallel compilation on the allocation quality we explored the default algorithm behavior under increasing register pressure. To this end, we experimented both with changing the number of available register and with the parallelization degree of the benchmarks.

Figure 33 presents the results. We can see that for most of the cases the algorithm required 40 to 44 machine registers to support up to 7 parallel threads without significant spilling. 32 machine registers, available in many RISC architectures, were enough to support up to 3 threads without spilling. Those numbers also correlate well with our manual estimation of the required number of machine registers for each of the cases. The almost constant horizontal distance between the curves effectively expresses the per-thread register requirements. The spill cost grows rapidly with the decrease in the number of registers as the allocator is forced to spill more heavily-used variables. Registers 28-32 do not participate in allocation, explaining the flat curve in that range. A similar behavior is observed for the simulator execution time, as shown in Figure 34.
Figure 33: Spill cost for different parallelization degree and number of registers

Figure 34: Dynamic performance degradation
4.3.5 Complexity Considerations

In this section we take a look at the complexity implications of the parallel code compilation on the graph coloring algorithm. It is reasonable to expect that parallel code compilation should be more resource-intensive than sequential compilation due to the increased complexity of parallel programs. In this section we measure the amount of work performed by the coloring algorithm as function of the problem size (the number of live ranges). We consider two stages of the algorithm: the initial build and the subsequent coloring stage.

In sequential case, the initial build stage dominates the algorithm run time [9]. We use the number of edges in the graph as the metric for the amount of work in this stage.

For sequential code, the number of interference edges is usually linear in the number of live ranges. In the parallel code, the number of edges approaches $O(n^2)$, where $n$ is the number of live ranges. Moreover, parallel code increases the total number of live ranges: $n_{\text{parallel}} = \#\text{threads} \times n_{\text{sequential}}$. As a result of both factors, the number of edges in the parallel case is considerably higher.

The coloring stage also usually behaves linearly in the sequential case [7]. For $s$ spill decisions, the algorithm requires $O(s \times n)$ time during a single pass. While $s$ is usually reasonably low in sequential code (for RISC architectures), it can become the dominant factor in parallel interference graph coloring.

Table 3 summarizes our measurements when using 32 machine machine registers for allocation.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># passes</th>
<th># nodes</th>
<th># edges</th>
<th># constrained (s)</th>
<th># evaluated</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>seq</td>
<td>par</td>
<td>seq</td>
<td>par</td>
<td>seq</td>
</tr>
<tr>
<td>twolf (1)</td>
<td>1</td>
<td>4</td>
<td>28</td>
<td>136</td>
<td>0</td>
</tr>
<tr>
<td>wip (2)</td>
<td>1</td>
<td>2</td>
<td>87</td>
<td>220</td>
<td>530</td>
</tr>
<tr>
<td>twolf (2)</td>
<td>1</td>
<td>6</td>
<td>88</td>
<td>338</td>
<td>697</td>
</tr>
<tr>
<td>wip (1)</td>
<td>1</td>
<td>1</td>
<td>100</td>
<td>117</td>
<td>984</td>
</tr>
<tr>
<td>bzip2 (1)</td>
<td>1</td>
<td>2</td>
<td>106</td>
<td>128</td>
<td>869</td>
</tr>
<tr>
<td>mcf</td>
<td>1</td>
<td>1</td>
<td>118</td>
<td>176</td>
<td>854</td>
</tr>
<tr>
<td>art</td>
<td>1</td>
<td>2</td>
<td>237</td>
<td>326</td>
<td>2148</td>
</tr>
<tr>
<td>bzip2 (2)</td>
<td>2</td>
<td>3</td>
<td>395</td>
<td>433</td>
<td>4434</td>
</tr>
<tr>
<td>crafty</td>
<td>2</td>
<td>4</td>
<td>2085</td>
<td>2098</td>
<td>30667</td>
</tr>
</tbody>
</table>

Table 3: Interference graph statistics

Each column in the table is divided into sequential and parallel sub-columns. The columns are:

- **# passes**: number of iterations the algorithm required to color the graph
- **# nodes**: number of live ranges comprising the interference graph
- **# edges**: number of interference edges in the graph
- **# constrained (s)**: total number of times there was a need to find a spill candidate
- **# evaluated**: total number of nodes evaluated while finding a spill candidate

Due to the nature of concurrent interference graph, the number of constrained nodes it contains is usually much higher than this number in the corresponding sequential code. This fact, along with higher number of iterations required to color concurrent interference graph, results in orders of magnitude higher number of calls to the spill candidate selection procedure – the non-linear part of the coloring stage.

To measure the number of evaluated nodes on a sequence of similar problems with increasing problem
size, we ran the algorithm on code examples with varying number of symmetrical threads. The results, shown in Figure 35, confirm our estimation regarding the $O(n^2)$ behavior of the coloring stage.

4.4 Optimal Register Allocation

Although graph coloring allocators were shown to give high quality allocations on machines with wide register files, they seem to give much worse results on more restricted allocation problems. For example, RISC architectures usually offer a highly regular register usage, small number of addressing modes and wide register files (32 registers of each type). CISC architectures such as x86, on the other hand, offer only a few registers with sometimes predefined roles, but provide much wider set of addressing modes. For such irregular architectures, techniques developed for graph coloring allocators are mostly inefficient and result in poorly optimized allocation with extensive spilling.

Recently, academic research in this area has turned to a completely different direction: optimal or near-optimal solution to register allocation. Instead of representing register allocation by a graph coloring abstraction, the newly proposed approaches reduce the problem to an integer-linear programing [3,43,44] or boolean satisfiability [32]. Other approaches attempt a near-optimal solution with iterative approximation such as Lagrangian Approximation [21]. Most of these approaches have quite expensive overhead of optimally solving the formulated problem, but provide much better allocation results for cases with extensive spilling.

In the rest of this section we describe some of the proposed solutions and propose theoretical approaches for adopting them to register allocation for parallel threads.

*Integer linear programming* (ILP) approach was proposed by Goodwin and Wilken [42] and later extended by Fu and Wilken [44] and by Kong and Wilken [43]. The idea is to represent register allocation as a set of *allocation decisions* and *allocation constraints*, so that an optimal register allocation (ORA) is

Figure 35: Number of evaluated nodes as function of the graph size

![](image)
represented as an assignment over *allocation decisions*. Both *allocation decisions* and *constraints* are then translated into an integer linear program – a set of integer variables and linear equations. The cost of an allocation is represented by a cost function over the decision variables. The algorithm then applies a generic ILP solver (using *simplex* algorithm [35]) to find an assignment to decision variables that minimizes the cost function.

The ILP model uses *single-symbolic constraint* to represent interference between virtual registers: at any given execution point a hardware register may be occupied by at most one virtual register. Symbolically this condition is expressed as:

\[ \forall e \in E, \sum_i (R_{i,x[e]}) \leq 1 \]

Where \( R_{i,x[e]} \) – is an expression over decision variables that receives value 1 if virtual register \( X \) is allocated to the hardware register \( i \) at execution point \( e \) and 0 otherwise.

The same modeling can be applied to our formulation of interfering registers in case of parallel code compilation. Instead of considering one program point at a time we need to consider a set of program points that may be executed concurrently. The same *single-symbolic constraint* for parallel code can be formulated as:

\[ \forall e_1 \in E, \sum_i (R_{i,x[e_1]}) + (\forall e_2 \parallel e_1, \sum_{x \in \gamma} (R_{i,y[e_2]})) \leq 1 \]

Adding above constraints should preserve correctness of the allocation for parallel code while the same claims on optimality still hold.

Number of new constraints is linear in number of parallel execution points. Although we don't have empirical numbers, we believe that adding them to ORA model should not affect the feasibility of this approach.

Appel and George [3] proposed to separate the problem of optimal register allocation into two phases: the problem of finding spill decisions and the problem of finding a register assignment that maximizes register aliasing. In the first phase an ILP solver is run to find optimal spilling such that at every execution point there is at most \( K \) registers occupied by the live virtual registers, and in the second phase those values are assigned with hardware registers. The second stage is guaranteed to succeed without further spilling if register-to-register move operations are introduced between code instructions (in some cases it would also require register-to-register exchange instructions such as \( xchg \), supported by x86).

Such partition is claimed to produce faster solution of spill decisions problem while enabling a choice between faster, but sub-optimal heuristic or optimal, but slower coalescing.

We consider this approach less suitable for register allocation for parallel code: according to the authors, heuristic solution to coalescing phase gives unsatisfactory results. Contrary to quite straight-forward extension to the approach proposed by Goodwin, finding an allocation for parallel code with the algorithm proposed by Appel and George would probably require more than trivial modification of the coalescing
phase.

4.5 Related Work

We consider the work “Balancing Register Allocation Across Threads for a Multithreaded Network Processor” [45] to be the closest to ours. In this work, Zhuang and Pande propose an algorithm for register allocation targeted at IXP network processor. Similarly to an Inthreads machine, this processor (each of its processing units) offers a large regular register file shared by all concurrently executing threads. Each processing unit of the IXP processor executes code of a single thread at a time, but it can switch to a different thread when encountering a long-latency instruction, such as a memory load or an IO access, or when an explicit context switch instruction is issued.

Formally, the allocation problem solved in [45] can be expressed as:

\[ Formulation \text{ in (1) holds, and } \forall e_1 \parallel e_2 \in CS, \forall v_1 \in \text{Live}(e_1), v_2 \in \text{Live}(e_2); \ Reg(v_1) \neq Reg(v_2) \]

(4)

Where:

- $e_1 \parallel e_2$  – denotes that $e_1$ may execute in parallel\(^1\) with $e_2$
- $CS$ – denotes the set of all context switch points ($CS \subseteq E$)

Note that register allocation problem, faced by [45], is a sub-problem of that, defined by the Inthreads architecture: the live ranges of different threads only conflict if they are live through some potential context switch point. Other live ranges, that do not cross context switch boundaries, can share the machine registers with other threads. The work in [45] doesn’t address the use of shared variables and the corresponding register allocation challenges.

The solution in [45] uses traditional graph coloring algorithm to initially color interference graphs of each of the parallel threads. Then it merges the interference graphs, performing recoloring if required. If number of colors is insufficient, the algorithm iteratively splits larger live ranges into less-constrained shorter live ranges.

In contrast, our solution creates a single interference graph, that represents all live ranges of all threads (also note that this graph is different from the one used in [45] due to the difference between formulations (4) and (2)). We then use traditional algorithm (including live range splitting) to color the graph according to the spill cost estimations. Our work also addresses the use of shared variables and spill code generation.

Sequential register allocation by graph coloring was discussed by [9,7,16,10,4,2]. Several approaches were used to evaluate and improve the heuristic function used by this algorithm: [6] proposed to use “best-of-three” approach, while in [40] genetic-programming was used to search the whole space of heuristic functions.

Recent years brought new approaches to the register allocation problem using different models that

\(^1\) With regard to the parallel execution model, provided by the context switching.
provide optimal or near-optimal solutions [21,43,44,32].

In [22] Koes compares heuristic graph coloring to optimal register allocation with a conclusion that heuristic provides results comparable to optimal graph coloring, and the advantage of the new approaches comes from a better modeling, rather than from the optimality of the solver.
5 Implementation Details

For our implementation we extended GCC compiler version 3.4.1 with Inthreads-specific features. The compiler was configured to use MIPS back-end with some modifications to make the output compatible with PISA architecture used by the SimpleScalar-based simulator. The set of optimizations was limited to -O1 (basic optimizations).

The syntax parser and the lexical tree representation were extended to support additional block attributes defined by the \#pragma inthread and Inthreads-specific operations such as INTH_START(), INTH_SET() and INTH_WAIT().

After parsing the source file, the lexical representation is translated into an extended RTL representation (Register Transfer Language). The RTL language is used as the base internal representation for the GCC optimizations. We used special marker instructions to represent a thread boundaries. We also added RTL instructions to represent the corresponding Inthreads machine instructions: \texttt{inth.start}, \texttt{inth.set}, \texttt{inth.wait} etc.

We modified the CFG construction algorithm to create a CCFG graph: this required special handling of the synchronization instructions during basic-block partitioning and generation of synchronization edges.

In addition, we implemented a set of Inthreads-specific analysis algorithms over the CCFG structure. Those include parallel data flow, shared variable recognition and concurrent interference detection.

GCC optimization steps such as dead code elimination, common sub-expression elimination, loop-invariant code motion and copy-propagation (pre- and post-register allocation) were modified to preserve the program semantics by special handling of shared variables at synchronization operations (as discussed in 3.5). There is also a number of CFG structure optimizations, such as if-conversions or basic block reordering, that required modifications to preserve consistency of the CCFG features, such as special basic block partitioning, synchronization edges, thread markers etc.

Our first implementation for register allocation was based on static partitioning of the register file, manually specified by the programmer. It used the GCC’s default register allocation algorithm with separate local and global allocation. This implementation turned to be problematic as it lacked proper support for sharing registers between threads and lacked compiler-driven balancing between parallel thread requirements.

In search for a generic solution we switched to using the graph coloring allocator (this allocator was removed from the more recent GCC versions). The graph coloring allocator was modified to support the register allocation for parallel code. This mainly affected the interference graph building algorithm, where Inthreads-specific steps were added to enable cross-thread live range building and parallel interference computations. The spill candidate selection procedure was modified to support interference graphs that consist solely from short live ranges (discussed in Section 4.3.4.3): in such a case the procedure chooses the live range with minimal cost (the default implementation would abort as such cases are not expected to
happen in sequential compilation).

We also modified the spilling algorithms to support controllable allocation policies. These modifications allowed us to experiment with different spill cost heuristics and run GP algorithm to evolve spill cost expressions.

Finally, the back-end part of the compiler was modified to translate RTL representation to Inthreads assembler.
6 Conclusions

In this work, we explored compilation methods for a rare type of a parallel architecture that allows for fine-grained parallel threads to run in a shared processor context. We presented several modifications to internal representation and traditional compilation steps that allow us to preserve the compilation correctness for parallel code.

We have presented an approach to register allocation for parallel threads. Our approach extends the well-known graph coloring paradigm from sequential to parallel code compilation. To do so, we build a parallel interference graph that represents both sequential and parallel interferences between different variables used by the program.

We address the correctness challenges that arise from applying the traditional allocation and spilling algorithms to parallel code. Some of those challenges, such as register coalescing and thread communication, are solved by correct construction of the parallel interference graph. Others, such as correctness of interference region spilling, are solved by modifying the corresponding algorithms to support the parallel code compilation.

We measured the performance of the modified graph coloring algorithm. Our measurements show that this paradigm performs reasonably well when applied to the parallel code.

One of the major concerns we present is the increase in the algorithm complexity. While applying graph coloring to sequential code often exhibits linear behavior, applying the same paradigm to parallel code exhibits quadratic behavior.

We should also note that the area of fine-grained parallelism as implemented by the Inthreads architecture presents many challenges and opportunities for compiler optimizations. Solving the function call limitation is one of the immediate challenges that we plan to address. Implementing an automatic parallelizing front-end is another such opportunity.

One could also consider combining register pressure information into a parallelizing compiler front-end to estimate the limits on available parallelism and make sure it produces code with feasible register allocation requirements. Our observations also show that there are opportunities for optimizations that could take further advantage of shared register file: per-thread register requirements can be effectively reduced by sharing read-only and pre-computed values.
Bibliography


[7] Preston Briggs, Register Allocation via Graph Coloring, Ph.D. Thesis, Rice University, April 1992


Optimization for Parallelizing Compilers, in 1992 Workshop on Languages and Compilers for Parallel Computing.


[22] David Koes, What’s Wrong with Graph Coloring?, Carnegie Mellon University, 2005


[31] Mayur Naik, Jens Palsberg, Correctness of ILP-based Register Allocation, August 2004

[32] Nachiketh R. Potlapally, Solving Register Allocation Using Boolean Satisfiability, Princeton University,
May 2003.
[37] Dennis Shasha, Mark Snir, Efficient and Correct Execution of Parallel Programs that Share Memory, in ACM Transactions on Programming Languages and Systems, April 1988
שיטות בקומפרליזציה של קוד ממוכבל
באירקיטוטורה ענ הקשה משותף

חיבת על מחקר

לשים مليית חלקי של הדרישות לקבלי התוכן מוגשת למעדין
במגדיע המחשב

גרגורי שקנובור

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

سمي תשמ"ז, חיפה יוני 2006
הערה תודה

המחוק עשה בנחתת פורפ' אסף שוסר בפקולטה למדעי המחשב, הטכניון – מכון טכנולוגי לישראל.
ברזון לבתא הערכה עמיקה לאמך ונתמך על סבלנות ואינטואיטיותionales מקצועיות.
<table>
<thead>
<tr>
<th>רישומת ציורים</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="Image 1" /></td>
</tr>
<tr>
<td><img src="#" alt="Image 3" /></td>
</tr>
<tr>
<td><img src="#" alt="Image 5" /></td>
</tr>
<tr>
<td><img src="#" alt="Image 7" /></td>
</tr>
<tr>
<td><img src="#" alt="Image 9" /></td>
</tr>
<tr>
<td><img src="#" alt="Image 11" /></td>
</tr>
<tr>
<td><img src="#" alt="Image 13" /></td>
</tr>
<tr>
<td><img src="#" alt="Image 15" /></td>
</tr>
<tr>
<td><img src="#" alt="Image 17" /></td>
</tr>
<tr>
<td><img src="#" alt="Image 19" /></td>
</tr>
<tr>
<td><img src="#" alt="Image 21" /></td>
</tr>
<tr>
<td><img src="#" alt="Image 23" /></td>
</tr>
<tr>
<td><img src="#" alt="Image 25" /></td>
</tr>
<tr>
<td><img src="#" alt="Image 27" /></td>
</tr>
<tr>
<td><img src="#" alt="Image 29" /></td>
</tr>
<tr>
<td><img src="#" alt="Image 31" /></td>
</tr>
<tr>
<td><img src="#" alt="Image 33" /></td>
</tr>
<tr>
<td><img src="#" alt="Image 35" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>רישומת טבלאות</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="Table 1" /></td>
</tr>
<tr>
<td><img src="#" alt="Table 3" /></td>
</tr>
</tbody>
</table>
תקציר

babudah aha anavon hokrim shonot komplikiletz la-arithetrov mokhibiyt su hakher b'aynu meshu.l

Arithetrov ha maspekhat ma'amirat levati mokhibiyt b'remam meshudah. Lhadil midribra arithetrov.

Arithetrov ha midribra ato hakrim hakher shonot: hakrim meshufot la kr ech hokrim ala go ato hakrim.

Ko'b aro'm. La go maspekhat sti she amayim sec'or ma'midra celim midriyot b'oro ye'sh la'meshu'im meshufot.

Nachem mashenesh kotemfilitza di-she'elati ca'em mokhibit a'thatecha maspek k'dek mokhib iti she.

Bhahshar maspek k'llak mokhib 'ui komplikiletz shel hakrim midvot lehakim ato komplikiletz.

Shel hakim mokhibit elo'utl AE'tamimizot 'tor shemara el benot hakid.

"La'hote mubadah mespek temporot ema'orim l'emada Had VeDarot hashemad, Vor mechibbit bermat

Pekdorah mokhata ilen ile halut aha kah mansho be'atra melah. Boszuk k'am, begehit Evarik ha rahutot

Zo're me'mbeyatem ale kr shemetatem ale ilen halut hashabor no'ahid b'iydlu be'ezui mansho. Bema'am

Halihbey ale begehit ile', arithetrov hidshor matonot lehut ye'el ile halut ale halut.

Hakrim mokhibit.

Arithetrov ha enshe la'athor lila'mit mekim bh' mechibbit bermat hakdorah ale maspek cy.

Lafeil b'mishmeh bahr mespocht be'Dakim bermat hakim (ILP) ilen halut ha rahutot.

Avison matin b'gel ato hakrim leshita va. Halut (SMT) alen matin b'gel hakrim leshita.

Ketam yis'mon hakim. Arithetrov ga midiv'a ato'ach celim] pekdot sec'or mofetot ale der'ili:

Zai ha mofetot mishu'im ato mishav ma'mud: hagamirim (hakimah am'merim mihdik) vor kokrim.

INTH_WAIT

Arithetrov mofetah la mofetah Release Consistency. hakdorah mishu'im ato mishav ma'mud:

Beyo'ah la mishu'im bispit shel hakrim midvot.

Shefet ha INTH-C ha mishu'mat she'fet kr'er b'it hakenet (am mokhibit ao'monit) l'hi komplikiletz.

Shefet ha INTH_START. INTH_SET, INTH_C

Mazridet rahotot s'nekulit hidamot levelelet hakenet 'ui arithetrov:

(b) INTH-C.

Menbour mofetahale s'hef kr'ayil lekomufilitzah k'dek mokhibit meshuf b'oro mofetah ale mishu'im bispit.

Arshe am mishu'im bayit b'mekhibit bermat mokhibit vor temayot ale benot. Avison 3 most ale mishu'im.

Ketam b'mokhibit, kefa la vorefahat b'se'it yonelaniyot k'dek, k'ilela bnet su lehelet yonel k'dek ato.

Likom k'ayit mishu'im ah voshef ha yonelaniyot shel INTH-C. Shefet ha mohor go ha ayneg lehaken ato al'orotah.

Arbutrac mokhibit b'komufilitzah kr'ayil.

Cdi leneh aha ha hakenet, komplikiletz shel hakrim mishu'im ato zom'ay b'mishu'im me'amorim 'akrim: yonel Pirim.

Ani'ayot (ma'asot ha'ayot hayish emorim) len'orin forpitah (ma'asot ha'ayot hayish emorim).

Yitizon mofetim komhibit b'komufilitzah ar'or yad berehah.

C CFG.

Beyo'ah k'dek mofetim komufilitzah mishu'mah

Yitizon komufilitzah lekomufilitzah ar'or yad berehah.

Concurrent CFG. Yitizon ha mishu'mah ale ha'ayot komhitem komufilitzah kashot sec'or (ar'or release

Hamahat shel yonel hayin ha. (4)
acquire (cdownatm) the lock whenever the lock is needed by all the threads. This is a necessary protection mechanism to ensure that the lock is acquired only by the thread that needs it.

The lock is acquired using the `pthread_mutex_lock` function, which takes a mutex as an argument. The mutex is a synchronization object that represents a lock that can be acquired and released by multiple threads.

The `cdownatm` function is used to acquire the lock for the current thread. It takes a null pointer as an argument and returns the address of the `cdownatm` function.

The lock is released using the `pthread_mutex_unlock` function, which takes a mutex as an argument.

In addition to the lock, the thread also needs to acquire a read lock on the shared data structure. This is done using the `pthread_rwlock_rdlock` function, which takes a reader writer lock as an argument.

The lock is released using the `pthread_rwlock_unlock` function, which takes a reader writer lock as an argument.

copy propagation (cpropagation)

Copy propagation is a compiler optimization technique that moves code that is never executed to a more optimal position in the code. This can improve the performance of the compiled code by reducing the amount of code that needs to be executed.

The copy propagation is performed by the compiler when it generates the final executable code. The compiler analyzes the code and identifies sections of code that can be moved to a more optimal position in the code. This can improve the performance of the compiled code by reducing the amount of code that needs to be executed.

In this case, the compiler has identified the copy propagation as a technique that can be used to optimize the code. The compiler has moved the code that is never executed to a more optimal position in the code.

The copy propagation is performed by the compiler when it generates the final executable code. The compiler analyzes the code and identifies sections of code that can be moved to a more optimal position in the code. This can improve the performance of the compiled code by reducing the amount of code that needs to be executed.

In this case, the compiler has identified the copy propagation as a technique that can be used to optimize the code. The compiler has moved the code that is never executed to a more optimal position in the code.

The copy propagation is performed by the compiler when it generates the final executable code. The compiler analyzes the code and identifies sections of code that can be moved to a more optimal position in the code. This can improve the performance of the compiled code by reducing the amount of code that needs to be executed.

In this case, the compiler has identified the copy propagation as a technique that can be used to optimize the code. The compiler has moved the code that is never executed to a more optimal position in the code.