Weld: Fast Data-Parallel Computation on Modern Hardware

by

James J. Thomas

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Master of Engineering in Computer Science and Engineering at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

© James J. Thomas, MMXVI. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Author

Department of Electrical Engineering and Computer Science

May 20, 2016

Certified by

Matei A. Zaharia
Assistant Professor
Thesis Supervisor

Accepted by

Christopher Terman
Chairman, Masters of Engineering Thesis Committee
Abstract

Modern hardware is difficult to use efficiently, requiring complex optimizations like vectorization, loop blocking and load balancing to get good performance. As a result, many widely used data processing systems fall well short of peak hardware performance. We have developed Weld, an intermediate language and runtime that can run data-parallel computations efficiently on modern hardware. The core of Weld is a novel intermediate language (IL) that is expressive enough to capture common data-parallel applications (e.g., SQL, graph analytics and machine learning) while being easy to parallelize on modern hardware, through the use of a simple “parallel builder” abstraction and nested parallel loops. Weld supports complex optimizations like vectorization and loop blocking, as well as a multicore CPU backend. Finally, Weld’s runtime can to optimize across library functions used in the same program, enabling further speedups that are not possible with today’s disjoint libraries. In this thesis, we describe the Weld IL and then turn to the multicore CPU backend, providing a theoretical analysis suggesting that it has low overheads and showing that microbenchmarks and real-word applications like TensorFlow have excellent multicore performance when ported to run on Weld.
Acknowledgments

First of all, I would like to thank my advisor Matei Zaharia. He has a unique combination of high-level technical insight and low-level expertise. I look forward to working with him in the future.

Next, I would like to thank my collaborators Shoumik Palkar, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Samuel Madden, and Saman Amarasinghe. Much of this thesis other than Chapter 5 is derived from a paper that we submitted together on Weld.

Thanks to Stephanie Wang for a helpful discussion regarding the material in section 5.4.

Finally, I want to express my gratitude to my family and friends for supporting me in this work.
Contents

1 Introduction ........................................... 9

2 Motivation for Weld ................................ 13
   2.1 Non-Goals ........................................... 16

3 Weld Intermediate Language ...................... 17
   3.1 Operators .......................................... 18
   3.2 Functional Sugar Operators ..................... 20
   3.3 Why Loops and Builders? ......................... 21

4 Implementation ..................................... 23
   4.1 Integration into Libraries ....................... 23

5 Multicore Backend .................................. 25
   5.1 Overview .......................................... 25
   5.2 Parallel Runtime Implementation .............. 26
   5.3 Multicore Builders ............................... 27
   5.4 Analysis of Parallel Runtime Overhead ....... 29

6 Evaluation ............................................ 35
   6.1 CPU Micro-benchmarks ......................... 35
   6.2 Existing Frameworks ............................. 37

7 Related Work ....................................... 39
Chapter 1

Introduction

Even though data-intensive computing systems are “embarrassingly parallel,” many of them perform poorly on modern parallel hardware. For example, a recent study shows that even in simple applications like PageRank, distributed graph frameworks can underperform a single core [26]. Likewise, in-memory databases can be an order of magnitude slower than hand written code [11]. We find similar 5–10× gaps from peak performance in machine learning libraries [1] and other systems [39].

One key problem is that utilizing modern hardware requires complex optimizations such as vectorization, loop blocking, and dynamic load balancing. These optimizations are difficult for compilers to implement [25] and must usually be built into each system by hand. Moreover, they vary widely across hardware platforms such as different CPU models or GPUs. At best, developers can leverage the work of experts by calling optimized kernel libraries like BLAS [23]. However, even these libraries cannot provide optimal performance because there is no optimization across library calls. The library approach works especially poorly for data-intensive applications, where simply writing results to memory between calls to library functions can dominate the processing cost.

To address this problem, we have developed Weld, an intermediate language and runtime that runs data-parallel applications efficiently on diverse hardware. Weld provides a restricted intermediate language (IL) that can express a wide range of parallel applications, including SQL, machine learning and graph analytics. It applies rich optimizations that are usually not feasible in more general compilers, and emits optimized code for each platform (Figure 1-1). Finally, Weld’s runtime provides a novel API that lets it optimize across libraries combined in the same program, unlocking significant optimizations that
Figure 1-1: Weld captures diverse data-parallel applications using a common intermediate language, and applies rich program optimizations to generate code for each hardware backend.

are not possible with today’s separately written kernels.

Weld’s IL captures parallelism through a novel representation based on nested parallel loops. These loops produce results through abstract data types called builders, which allow constructing a result in parallel without specifying a particular concurrency mechanism. Builders have restricted write-only, build-once semantics: reading a result is not allowed until all writes have finished, which simplifies both their implementation and analysis. Despite this restriction, we show that loops and builders can capture a wide range of analytics applications.

This loop-and-builder representation makes Weld amenable to optimizations that are difficult to apply in more general ILs like LLVM [22] and OpenCL [32], such as loop fusion, loop blocking, and data layout transformations. In addition, Weld’s design makes it portable onto diverse hardware, because parallel loops and write-only builders are concepts that most current parallel platforms can support.

Finally, getting high performance requires optimizing across functions in the same program. In today’s libraries, every kernel function is optimized in isolation. This creates significant inefficiency, especially in data-intensive code. For example, although TensorFlow [1] uses the Eigen library [17] for its operators, simply fusing adjacent operators can give a $16\times$ speedup in some programs by not materializing intermediate results between them. To address this problem, Weld introduces a runtime API based on lazy evaluation that lets it collect IL fragments from different libraries in a computation and optimizes across them. For example, if a user runs a math function on each row returned by a SQL query, Weld can vectorize calls to the
function across rows, or even perform loop blocking, a change that would not make sense in either library on its own. To our knowledge, Weld is the first system to perform such optimizations across libraries.

We focus in this thesis firstly on describing the Weld IL and secondly on presenting and evaluating the Weld backend for multicore x86 CPUs. While some of the multicore backend’s design is inspired by prior work, we present a novel analysis that shows it has low overheads despite supporting very fine-grained load balancing. On SQL, machine learning and graph benchmarks, the multicore backend is competitive with hand-tuned code and matches or outperforms optimized systems such as the HyPer [27] database and the GraphMat [34] graph framework. In addition, we have integrated Weld into the TensorFlow machine learning framework, and the multicore backend outperforms vanilla TensorFlow on a common algorithm and is competitive with handwritten code.
Chapter 2

Motivation for Weld

The main goal of Weld is to provide high performance for data-parallel applications on modern hardware. Today, this task requires significant manual effort in each system. In this section, we describe the two features of Weld that enable this goal: a new IL and a runtime that allows rich optimization across processing functions.

New Intermediate Language. The key enabler of performance in Weld is its new IL tailored to data-parallel computation. We chose to develop a new IL for two reasons: 1) to make it amenable to aggressive optimizations for parallel code, and 2) to allow expressing computations from different libraries in a common format.

To understand the motivation for an IL, it is useful to consider current ILs such as LLVM [22], OpenCL [32] and SPIR [20], which are used in existing compilers. Most of these ILs were developed to optimize single-threaded code (or code on one GPU core), and operate on low-level constructs such as pointers and shared memory. This makes it difficult for compilers to identify and transform higher-level program structures and to leverage parallel hardware features such as SIMD instructions and multicores. For example, C compilers often fail to take advantage of vector instructions [25], and systems that try to identify parallel structure from sequential code, such as Polly [16], are generally only viable for programs with simple control flow and affine memory accesses. In contrast, Weld provides explicitly parallel constructs, such as loops and builders, over collections of compound data types. This higher level of abstraction allows the system to directly leverage many types of parallel hardware.

In practice, data processing frameworks can integrate Weld in several ways. Many frameworks, such
as relational databases and TensorFlow [1], break down computation into a small set of operators (e.g., physical operators in a database). In these cases, writing these operators in Weld will automatically allow both fast performance for each operator and optimization across them. Some systems already perform runtime code generation to a language such as C [27, 2, 19]; these can be modified to emit Weld, as we show in our evaluation.

**Runtime.** The Weld runtime enables rich optimizations across data processing functions and libraries. As applications call Weld-enabled functions, Weld’s API uses lazy evaluation to build a single IL program for the specific combination of functions used, and executes them only when a result is forced. Before executing the code, Weld applies a large library of transformations, including loop fusion, loop blocking, data layout changes, and vectorization. Weld’s loop and builder based IL is designed to easily support such transformations, allowing different back-ends to optimize the code for each platform.

As motivated in the introduction, cross-function optimization is especially important in data-intensive workloads, where the data movement between different processing functions can dominate execution time. With today’s compilers and ILs, such optimization does not usually happen. The code representation of each library is too low-level for compilers to optimize across them—indeed, many compilers do not even analyze the constructs used for parallelism in various libraries (e.g., thread creation and atomic instructions) and so they cannot optimize across parallel libraries. Weld thus unlocks significant optimizations that are not possible with today’s libraries.

**Example.** To illustrate the benefits of Weld, we compared the performance of a TPC-H [35] query written by hand in several languages (Java, Python, and C), two databases (MonetDB and HyPer), and Weld. Figure 2-1 shows the results. We started by implementing the query as a simple loop over column-oriented data in Java, Python, and C. Unfortunately, none of them emit optimal code for this query—manually vectorizing the C code using Intel AVX intrinsics led to a $2.7 \times$ speedup over the loop in C. MonetDB [6] is an analytical database that calls optimized kernels for each of its operators, but it incurs significant overhead due to the data copies between these operators. Finally, HyPer [27] compiles SQL queries to LLVM and comes much closer to our loop in C. Even in this case, HyPer was not designed to use vector instructions, and LLVM cannot automatically vectorize its generated code. In contrast, Weld directly generates vectorized code from a short IL program for the query.
Figure 2-1: Performance of TPC-H Query 6, written as a loop in several languages and running on the MonetDB [6] and HyPer databases. Results are single threaded at scale factor 10.

Figure 2-2: TPC-H query 6 optimized in Weld.

Figure 2-2 further shows how we achieved high performance in Weld through rich program optimizations. We started with a row-oriented version of the SQL query, and applied a generic transformation that converts row-oriented data structures (“array of structs”) into column-oriented ones. Weld then matched the unvectorized loop in C. Finally, we applied Weld’s vectorization transformation. Both of the transformations here act directly on the loop/builder IL and thus work on general Weld programs.
2.1 Non-Goals

Weld has several non-goals. First, Weld does not currently attempt to be a full blown “human friendly” programming language; rather, as an IL, Weld is meant to be generated using higher level libraries. It would be interesting to compile subsets of existing languages to Weld [31], but for now we have focused on designing an expressive IL that supports rich optimizations.

Second, Weld does not target distributed environments, focusing instead on performance on a single machine. Extending a single IL to support distributed hardware would create tradeoffs: for example, Weld currently allows random access into arrays, which is necessary in many fast single-machine algorithms but would be inefficient in a cluster. Developers can integrate Weld into distributed systems, however, as we show with Spark SQL.
Chapter 3

Weld Intermediate Language

Weld’s IL is a small, statically typed language that supports scalar expressions and two parallel constructs: loops and builders. It has some similarity to functional languages, in that variables are immutable once defined (the IL uses Static Single Assignment form) and some of its operators take functions as arguments. However, Weld is not truly functional, because functions are not first-class values (they cannot be assigned to variables), and recursion is disallowed. As a result, Weld programs have a simple, finite call tree that is known at compile time.

**Listing 3.1:** A simple Weld program to add two integers.

\[(x: \text{int}, y: \text{int}) \Rightarrow x + y\]

Weld supports several data types, shown in Table 3.1. Apart from scalars, there are three collection types: structures, variable-length vectors, and dictionaries. We chose these types because they appear commonly in data-intensive applications as well as low-level data processing code (e.g., dictionaries are useful for database joins). These types can be nested to represent more complex data.

<table>
<thead>
<tr>
<th>Primitive Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scalars</strong></td>
</tr>
<tr>
<td>char, int, long, float, ...</td>
</tr>
<tr>
<td><strong>Structures</strong></td>
</tr>
<tr>
<td>{T1, T2, ...} for field types T1, T2, ...</td>
</tr>
<tr>
<td><strong>Vectors</strong></td>
</tr>
<tr>
<td>vec[T]</td>
</tr>
<tr>
<td><strong>Dictionaries</strong></td>
</tr>
<tr>
<td>dict[K,V]</td>
</tr>
</tbody>
</table>

**Table 3.1:** Primitive data types in Weld.
### Core Operators in Weld

#### Basic Operations
- **Arithmetic expressions:** \( a + b, -a, a \& b, \) etc.
- **Let expression:** `name := expr; body`, to assign variable names
- **if:** `(condition) on_true else on_false`
- **while:** `(init, update): sequentially loops update function starting from an initial value`

#### Collection Operations
- `lookup(vector, index)` and `lookup(dictionary, key)`
- `struct.0, struct.1, etc.` to access fields in a struct
- `len(vector)` and `len(dictionary)`
- `sort(vector, keyfunc)` to sort a vector based on the key returned by `keyFunc`
- `tovec(dictionary)` to get entries of a dictionary as a `vec[K,V]`

#### Builder Operations
- `merge(builder, value)` to merge values into builders
- `result(builder)` to get a result from a builder
- `for(vector, builders, func)` to loop over a vector in parallel (§3.1)

<table>
<thead>
<tr>
<th>Builder Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vecbuilder[T]</code></td>
<td>Builds a <code>vec[T]</code> from merged values of type <code>T</code></td>
</tr>
<tr>
<td><code>merger[T,func,id]</code></td>
<td>Builds a value of type <code>T</code> by merging values with a commutative function <code>func</code> and an identity value <code>id</code></td>
</tr>
<tr>
<td><code>dictmerger[K,V,func]</code></td>
<td>Builds a <code>dict[K,V]</code> by merging values with a commutative function <code>func</code></td>
</tr>
<tr>
<td><code>vecmerger[T,func]</code></td>
<td>Builds a <code>vec[T]</code> by merging values of type <code>{index,T}</code></td>
</tr>
<tr>
<td><code>groupbuilder[K,V]</code></td>
<td>Builds a <code>dict[K,vec[V]]</code> from merged values of type <code>{K,V}</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3.2: Expressions in the core Weld language.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Table 3.3: Builder types in Weld.</th>
</tr>
</thead>
</table>

### 3.1 Operators

Table 3.2 lists the core operators in Weld. Weld contains basic operators for arithmetic, assigning names to values, sequential looping, and common operations on collections (e.g., `lookup` for random reads into a vector).

The unique part of Weld is its parallel constructs. Weld has two such constructs: a **parallel loop** to iterate over data, and **builders** for constructing results. A builder is an abstract data type that computes a result in parallel. Builders are write-only, build-once: expressions such as parallel loops can merge values
into the builder, but a final result can only be materialized once, after the merges are done. Weld includes multiple types of builders, as shown in Table 3.3. For example, a `vecbuilder[T]` takes values of type `T` and builds a vector of the merged values. Meanwhile, a `merger[T,func,id]` takes a commutative function and an identity value and combines `T` values into one result.

Builders support three basic operations. `merge(b, v)` adds a new value `v` into the builder `b` and returns a new builder¹ to represent the result. Merges into builders are associative, enabling their arbitrary reordering (builders which take functions as part of their type must also have the functions be commutative). `result(builder)` destroys the builder and returns its final result: no further operations are allowed on it after this. Finally, `for(vector, builders, func)` applies a function of type `(builders, T) => builders` to each element of a vector in parallel, updating one more builders for each one, and returns the final set of builders. This is the only operator to launch parallel work in Weld.

Listing 3.2: Some simple examples of using builders.

```plaintext
// Merging two values into a builder
b1 := vecbuilder[int];
b2 := merge(b1, 5);
b3 := merge(b2, 6);
result(b3) // returns [5, 6]

// Using a for loop to merge multiple values
b1 := vecbuilder[int];
b2 := for([1, 2, 3], b1,
    (b, x) => merge(b, x + 1));
result(b2) // returns [2, 3, 4]

// Merging results only for some iterations
result(
    for([1, 2, 3],
        vecbuilder[int],
        (b, x) => if(x > 1) merge(b, x) else b
    ) // returns [2, 3]
)

¹In practice, some mutable state will be updated with the merged value, but Weld’s IL treats all values as immutable, and so we represent the result as a new builder object in the IL.
<table>
<thead>
<tr>
<th>Sugar Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(v: vec[T], f: T=&gt;U): vec[U]</code></td>
</tr>
<tr>
<td><code>filter(v: vec[T], f: T=&gt;bool): vec[T]</code></td>
</tr>
<tr>
<td><code>reduce(v: vec[T], id: T, f: (T,T)=&gt;T): T</code></td>
</tr>
<tr>
<td><code>zip(v1: vec[T1], v2:...) : vec[{T1,T2,...}]</code></td>
</tr>
</tbody>
</table>

Table 3.4: Some of the functional sugar operators in Weld. Each operator is mapped to for loops and builders.

Note that because `for` itself returns a builder, it can be nested and composed. This lets Weld express nested parallel programs, including irregular parallelism (where instances of the inner loop do different amounts of work). It also makes Weld amenable to a wide range of loop transformations such as blocking and loop fusion.

Listing 3.3: Using nested loops to compute a result over nested data. Here we flatten some nested lists.

```plaintext
code
lists := [[1,2],[3,4,5],[6]];
result(
  for(lists, vecbuilder[int], (b, list) => for(list, b, (bl, elem) => merge(bl, elem)))
) // returns [1,2,3,4,5,6]
```

Finally, Weld places two restrictions on the use of builders for efficiency. First, each builder must be consumed (passed to an operator) exactly once per control path, to prevent having multiple values derive from the same builder, which would require copying its state. Formally, builders are a linear type [37]. Second, functions passed to for must return builders derived from their arguments, and not call result on them. These restrictions let backends safely implement builders using mutable state.

### 3.2 Functional Sugar Operators

For convenience, Weld also contains higher level syntactic sugar for commonly used functional operators such as `map` (Table 3.4). These sugar operators act as macros and map directly into loops and builders before the program is processed further. However, we found them useful to include to facilitate code generation from libraries.
For example, the last two snippets in Listing 3.2 implement a map and filter respectively. Simply switching these loops to use a merger can implement a reduce.

### 3.3 Why Loops and Builders?

Weld contains only loops and builders as its core constructs, and uses them to implement higher level sugar operators. A strawman design for a parallel IL might include higher level functional operators as part of the core language. Unfortunately, this design prevents many optimizations from being expressed easily. Consider the example in Listing 3.4, where two operations produce a result over the same input data:

Listing 3.4: A map and reduce over the same input data.

```java
data := [1,2,3];

r1 := map(data, x => x+1);

r2 := reduce(data, 0, (x, y) => x+y)
```

Even though both the map and the reduce operations could be computed in a single pass over the data, no operator akin to mapAndReduce exists which would compute both values in one pass. While some systems based on functional operators [15] can pattern match on combinations of operators and produce fused programs for some of them, no general system exists to capture such optimizations for any combination of operators. More complex optimizations, such as loop blocking, would be even more difficult to express in a functional IL. The same holds true for ILs based on relational algebra operators, such as LINQ [31].

By exposing all parallelism through a single loop construct over builders, patterns like the above can easily be fused into programs such as Listing 3.5.

Listing 3.5: for operating over multiple builders to produce both a vector and an aggregate in one pass.

```java
data := [1,2,3];

result(
    for(data,
        {vecbuilder[int], merger[+]},
        (bs, x) =>
            {merge(bs.0, x+1), merge(bs.1, x)}
    )
) // returns {[2,3,4], 6}
```
The other restrictions of Weld also let the compiler easily apply more types of optimizations compared to more general frameworks such as LLVM and OpenCL. A known call tree allows transformations on the AST without having to worry about unresolvable control paths. The write-only, build-once property of builders allows backends to generate highly efficient implementations of these types for each hardware platform (e.g., by keeping local state in each thread without synchronization until a call to `result`). Finally, keeping the IL small simplifies both backends and optimization passes.
Chapter 4

Implementation

Weld’s parser, transformation library and code generation are implemented in Scala. The CPU code generator emits LLVM IR for every core operator in the language. We have implemented the builder data types in C, which we compile to LLVM bitcode and link against the generated LLVM IR. We send the LLVM IR generated for a Weld program to a JIT that assembles the IR and loads the executable into the current process, at which point we can call it.

4.1 Integration into Libraries

TensorFlow. Our integration with TensorFlow required a graph rewriter that converts a TensorFlow operator graph into a Weld program (279 lines of Python). Each operator maps to a partial Weld program which are fused before execution. We found TensorFlow’s existing support for used-defined operators sufficient to integrate dispatch of Weld code from the TensorFlow execution engine—that is, we did not need to make any changes to the core TensorFlow engine. We note that we only support a small subset of the ops TensorFlow currently provides.

Spark SQL. We augmented Spark SQL’s [2] existing Java code generator to produce Weld code to speed up the processing of individual partitions in a distributed dataset. The code generator analyzes a physical query plan and JIT compiles it. We supported three key operators: project, filter, and aggregate (with and without keys). Spark SQL’s existing Java code generator must take pains to generate code with optimizations like loop fusion already present, since the Java compiler cannot perform these optimizations automatically. In contrast, our Weld code generator took the simple approach of generating a separate
loop for each operator. Weld’s optimizations automatically fused the loops.

SparkSQL is able to read data in the Parquet [30] columnar format into off-Java heap buffers that are easily accessible by native code. To encode and decode data, we simply pass pointers to this data between Weld and the Java runtime. Later stages in the Spark processing pipeline that do not use Weld already supported reading off-heap data, so this feature was easy to integrate.
Chapter 5

Multicore Backend

Transformations on the IL lead to an efficient representation of a parallel program. Backends enable the realization of this representation by generating code for each Weld operator and builder. We now describe the implementation of the backend for multicore CPUs.

5.1 Overview

Because Weld’s core operator – the `for` loop – is by definition fully parallel, the job of the multithreaded backend is simply to schedule loop execution on multiple cores. The main challenges are (1) low-overhead dynamic load balancing across cores to ensure maximum throughput for loops with irregular load distributions (common in e.g., graph applications), and (2) synchronization-avoiding state construction with parallel builders.

A Weld program can be represented as a linear task graph, with each outer (`i.e., not nested`) `for` loop represented as a single task that is a dependency of all loops and statements that come after it. A loop can only execute after its dependencies.

Our parallel runtime system is inspired by the work of [36]. It creates a worker for each core and a task queue for each worker. A task simply consists of a range of iterations in a loop. Load is balanced by splitting loops across workers; a worker steals a task from the queue of a random other worker when it is idle and its own queue is empty. Initially, all iterations of the first loop in the program are given to the first worker. If an executing worker observes that its task queue is empty (a cheap check that happens every 64 iterations for innermost loops and every iteration for other loops), it creates a task consisting of half of the remaining iterations in the currently executing loop and adds it to its queue. Thus, task creation is
lazy in that new tasks are created only when they are likely to be stolen, limiting task creation overhead.

Tasks for nested loops are created by splitting the outermost loop with more than one iteration remaining. This policy ensures that expensive inner loops will be split across cores, but small, common-case inner loops will not incur task creation overhead.

Our parallel builders are implemented using standard shared memory data structure techniques, including maintaining per-thread copies of data structures, cache line padding to reduce false sharing, and others. Much of our flexibility in the parallel builder design, in particular our ability to maintain per-thread data structures until a result call, is due to the write-only, build-once specification of the builder contract (§3.1).

5.2 Parallel Runtime Implementation

Most of the logic for the parallel runtime, including the code to create the worker threads, initialize the task queues, and create new tasks, is in a C library. Each worker thread runs in a loop where it executes tasks from its task queue until the queue is empty, at which point it steals a task from another worker’s queue and places it on its own queue. To execute a task, a worker calls into the generated LLVM IR for the Weld program with the loop bounds specified in the task. Loops in the generated LLVM are augmented with code to check the worker’s queue every 64 iterations in innermost loops and every iteration in other loops. This check is cheap and involves just a read of a variable from memory.

If a worker’s queue is observed to be empty, all the variables in the Weld program as well as unnamed expressions that must be shared across task boundaries (e.g., a new builder expression in the arguments to a for loop) are saved in the current task structure, and a call is made into the C library that splits the current task into two tasks with the same saved variables and places the second one on the worker’s queue. When a worker calls into the generated LLVM to execute a task, there is a prologue that unloads all of the saved variables from the task for use in the LLVM.

We encountered one major challenge in the implementation of the parallel runtime. In a Weld program’s LLVM IR, many IR-level registers are touched in every loop, since in the case where the queue is observed to be empty all Weld-level variables must be saved into the current task. This confuses LLVM’s register allocation pass, which no longer reliably allocates the frequently accessed IR-level registers to machine registers. We investigated a few simple solutions, including flagging blocks that
save program variables as unlikely, but they did not change the register allocator’s behavior. Fortunately, the programs used in the evaluation do not have too many variables, so the register allocator still makes reasonably good decisions. For larger programs, however, the best solution seems to be to write a custom LLVM register allocation pass that understands the variable saving blocks and only allocates IR-level registers that are used outside of these blocks to machine registers.

5.3 Multicore Builders

We now review the design of the multicore builders. We focus on the multicore merger, vecbuilder, and dictmerger, finally describing an optimization that applies to all of the multicore builders. All of the builders are implemented with C libraries that are called from the LLVM IR for a Weld program.

The multicore merger is implemented with per-worker copies, with the copies laid out so that they do not share any cache lines, limiting false sharing. Workers merge updates into their copies, and when result is called on the merger, the per-worker copies are aggregated. This design is possible because the merger update function must be both commutative and associative. Commutativity means that even though a worker may execute the iterations of a loop out of order (i.e., if it steals iterations of the same loop from another worker), it can merge updates into its local copy as they are encountered without worrying about ordering. Associativity enables the final aggregation of the per-worker copies. In contrast, Cilk reducers [24] assume only associative update functions, adding implementation challenges and overheads related to preserving ordering.

The vecbuilder is also implemented with per-worker copies, but since merges into the vecbuilder are only associative (the final output must be in the same order as in a serial execution), some extra care is required to maintain ordering. In particular, when a worker executes a task that uses a vecbuilder, it creates a local vecbuilder, performs merges into it, and at the end of the task adds the local vecbuilder along with the task’s position in the program loop nest to its per-worker vecbuilder copy, which is simply a list of such (local vecbuilder, position) pairs. When result is called on the vecbuilder, the per-worker lists are combined and the combined list is sorted on the position field. The final vec is created by laying out the contents of the local vecbuilder’s in the sorted order. This vecbuilder makes merges fast and synchronization-free, at the expense of some extra (parallelizable, although we currently perform it serially) work when calling result.
Finally, the multicore `dictmerger` implementation is still a prototype, but is also based on per-worker copies that are merged together when `result` is called. Once again, this is possible because the update function for each key is associative and commutative. While this design was fine for our evaluation experiments, it can potentially see a space blowup of \( P \) (the number of workers) compared to a serial execution if all workers touch the full set of keys. This could be problematic if there is a large number of distinct keys.

There are a number of alternative parallel hash table designs that we would like to explore. One would involve fixed-size per-worker `dictmerger`’s – once a per-worker `dictmerger` reached some number of keys, merges for other keys would be sent to a global `dictmerger`, requiring some synchronization. If the key-value set to be merged is dominated by common keys, it is likely that the per-worker `dictmerger`’s would contain those keys, so merges to the global `dictmerger` would be relatively infrequent and would also involve uncommon keys, reducing the likelihood of contention (the global `dictmerger` would use fine-grained locking). Setting the size of the per-worker `dictmerger`’s involves a tradeoff between parallel space blowup and merges into the global `dictmerger`.

It should be possible to design multicore `vecmerger`’s and `groupbuilder`’s in a fashion similar to the multicore `dictmerger`, since both involve merges of key-value pairs into a dictionary-like structure. Each value in the `groupbuilder` is effectively a `vecbuilder` since it must maintain the serial ordering of updates, so some elements of the multicore `vecbuilder` are likely to be required for the multicore `groupbuilder`.

Finally, there is one important optimization that applies to all multicore builders. It generally takes a small amount of work to initialize a multicore builder (some calls to `malloc` to allocate the per-worker copies, etc.), and it is important that this work is amortized over a sufficient number of loop iterations. In particular, if there is a frequently executed inner loop that tends to have few iterations and uses a builder, it is important that the cost of multicore initialization is not paid for this builder unless its particular loop instance is actually split across multiple workers. Thus, for every builder in a Weld program, we initially create a single-threaded version. When a task split occurs, all single-threaded builders in the task that have not yet been materialized with `result` are converted to their multicore counterparts. Since most loops with few iterations are not split, their builders are never converted to multicore versions and overheads are limited.
5.4 Analysis of Parallel Runtime Overhead

As described above, most of the extra logic added for multithreaded execution (besides the cheap queue checks) runs in the task creation path. So our objective here is to show that task creations are few and therefore that the overhead from the parallel runtime system is low.

We will first show the expected number of steals in a Weld execution, adapting an argument made for the Cilk runtime system. The result for Cilk is that a Cilk computation with span (critical path length) $T_{\infty}$ executed on $P$ processors by Cilk's randomized work stealing scheduler experiences $O(PT_{\infty})$ expected steals [5].

We start by establishing a parallel between the execution of `cilk_for` loops and `for` loops under our parallel runtime, demonstrating that the $O(PT_{\infty})$ bound on expected steals holds for us as well. We can construct a corresponding `cilk_for` program from a Weld program by replacing each of the `for` loops with a `cilk_for` loop that performs the same work. A `cilk_for` loop is a parallel loop that is executed in a recursive divide-and-conquer fashion, with a function that recursively spawns itself to execute the first half of the input iterations and recursively calls itself to execute the second half. Once the number of input iterations falls below some grain size, the function actually executes the iterations. We will assume this grain size is 1, since grain size does not affect the steal bound and this grain size mirrors the behavior of our runtime system. Successive `cilk_for` loops must execute sequentially, just as in Weld.

The span of a `cilk_for` loop with $N$ iterations is given by $log(N) + \max_{i} T_{\infty,i}$, where the second term in the sum is the span of the loop iteration with largest span. This is because the critical path length to any of the leaves (individual iterations) in the parallel divide-and-conquer tree is the height of the tree ($log(N)$), so the iteration with largest span determines the overall critical path length. The span of a serial composition of `cilk_for` loops is the sum of the spans of the loops. Thus, for a program whose only non-$O(1)$ operations are potentially nested `cilk_for` loops – similar to a Weld program – the span is given by a sum of logarithmic terms.

Now we consider the state of the per-worker task queues in Cilk as a `cilk_for` program executes. Random work stealing occurs from the heads of work queues in Cilk. When a spawned function call is encountered, the continuation of the spawn is appended to the tail of the worker's queue, and the worker begins to execute the spawn. In the case of `cilk_for`'s, continuations consist of the second half of
the iterations passed to an invocation of the loop function. Thus, as a worker proceeds down the leftmost root-leaf path of the recursion tree of a nested \texttt{cilk_for}, the work queue grows with tasks consisting of earlier and earlier iterations of the outermost loop, and then a similar process repeats for the first nested loop in the first iteration of the outermost loop, and so on. Finally, when a worker finishes executing a task, it attempts to remove and execute the task at the tail of its queue, which will itself create more tasks that are appended to the tail. Precisely, when executing a particular iteration of a loop in a loop nest, if we consider the subtree of the divide-and-conquer recursion tree rooted at the least common ancestor of all iterations remaining in the worker’s task queue, the queue consists of one task representing the right branch (second half of the iterations) for every left branch taken in the path from the root to the iteration. Tasks closer to the root are closer to the head of the queue.

The thief-visible state of the single-element task queues in our runtime system is similar. In particular, in both runtime systems, the head of the queue always contains the latter portion of the iterations remaining in the outermost loop in the nest with more than one iteration left. There is one important difference: the actual range of iterations at the head of the queue. Consider a loop with no nested loops and 100 iterations that is initially assigned to some worker. In both runtime systems, the first task at the head of the worker’s queue will consist of the last 50 iterations. Now suppose that this task is stolen by another worker. In the \texttt{cilk_for} case, the new task at the head of the queue will consist of iterations 25 through 49, inclusive. However, in our case, the lazily created task will consist of the latter half of whatever range of iterations is remaining for the worker to execute. So, if the worker has already completed the first 4 iterations of the loop, the new task will consist of iterations 27 through 49.

Despite this complication, we can still show an equivalence between a Weld program and a Cilk program whose span is bounded by the span of the \texttt{cilk_for} program corresponding to the Weld program. First, we consider any execution of a Weld program and all of the task creations that occur during it. We construct an equivalent Cilk program with explicit divide-and-conquer recursion that splits loops exactly as in the Weld execution. In particular, for every task split that occurred in the Weld execution, we split the corresponding task in Cilk program into three tasks. The first new task consists of the iterations of the parent task that were already executed by the Weld worker. The second task consists of the first half of the unexecuted iterations, which the Weld worker proceeded to execute, and
the third task consists of the second half of the unexecuted iterations, which the Weld worker placed on its task queue. The last two of these child tasks are clearly no more than half the size of their parent task. More precisely, if the parent task has two or more iterations of its original outermost loop left to execute, these last two child tasks will cover less than half this many iterations of the same loop. If the parent has only one iteration left to execute, it must have a nested loop, since otherwise it would not have been split. In this case, the last two child tasks will each consist of less than half of the iterations of the nested loop the parent task was executing. In both cases, the last two child tasks will consist of at most as many iterations as they would have had had they been the tasks created from the parent task by the standard cilk_for division process. Finally, the first of the above child tasks can be as large as the parent task, but we have complete freedom in how we split it in the Cilk program down to tasks of one iteration of an innermost loop, since these splits never occurred in the Weld execution. We will split this task in the standard cilk_for fashion, repeatedly into two equal pieces. There is guaranteed to be a corresponding task (one that covers the same range of iterations) in the Cilk program for every task in the Weld execution because the Cilk program performs exactly the splits in the Weld execution until it must split a task consisting of iterations already executed in the Weld execution (the case of the first child task above), a task that never appears in the Weld execution anyway. To ensure that the equivalent Cilk program has an execution where its task queue heads are always identical to those in the Weld execution, the Cilk program can first spawn a function that runs the first two of the above child tasks and then call a function for third child task, so that the third child is in the continuation and is pushed onto the queue.

We now show that for any task representing a single innermost loop iteration in the equivalent Cilk program, its depth in the recursive divide-and-conquer tree is no higher than the depth of the corresponding iteration in the regular cilk_for program plus one, meaning that the span of the equivalent Cilk program is the same as that of the cilk_for program, $O(T_\infty)$. Consider the recursion tree path from the root task to any single-iteration task. At some point in the path, the task will become an “Weld-executed” task (the case of the first of the three child tasks above), at which point it is divided exactly as in a cilk_for program down into tasks of one innermost iteration. Prior to this point in the path, each step cuts the task size by at least as much as in cilk_for program, as argued above. Finally, at the point in the path when the task becomes a Weld-executed task, we may have one extra step where the task size shrinks
by less than in the \texttt{cilk\_for} program (but does not grow). In the worst case, then, the tree depth of any single-iteration task in the equivalent Cilk program is at most one higher than that of its counterpart in the \texttt{cilk\_for} program, establishing the spans of these two programs as asymptotically equivalent.

Thus, however a Weld program executes, we can construct an equivalent Cilk program with an execution involving the same steals, identical thief-visible tasks at the heads of its task queues, and therefore identical program fragments assigned to each worker at any given time. This Cilk program always has span $O(T_\infty)$, so the Weld program should have $O(PT_\infty)$ expected steals. To see this more clearly, imagine that the Cilk execution determines the steals in the Weld execution, rather than the other way around. The assignment of program fragments to workers and the worker queue heads are identical at the start of the Cilk and Weld executions. Cilk and Weld workers execute the iterations of their identical program fragments in exactly the same order, and we can assume they make progress on identical program fragments at the same rate. The Cilk and Weld runtimes both perform randomized work stealing, and since whether and what a worker steals is dependent only on the program fragment left for the worker to execute and the task queue heads of the other workers, there is a one-to-one correspondence between the possible steals in the Cilk and Weld executions at any point in time. This invariant is maintained even as steals occur, since we can lazily determine the new head of a stolen-from queue in the Cilk execution based on the Weld work-splitting policy and the progress the worker has made in executing its now reduced program fragment. It is safe to determine this new queue head lazily even in the Cilk execution because the task breakdown below a queue head has no effect on steals or the worker’s execution of its program fragment, so a task only needs to be specified once it is actually exposed at the head. We can conclude that there is a one-to-one correspondence between possible executions in this Cilk model with lazy task creations and possible Weld executions, and the expected number of steals across the possible Cilk executions is $O(PT_\infty)$ since regardless of the way tasks are lazily created the Cilk program’s span is $O(T_\infty)$. The proof of the steal bound in \cite{5} depends only on the span of the Cilk computation and not on the particular sizes of the created tasks, so the proof still holds under the modified Cilk model. So the expected number of steals in a Weld execution is $O(PT_\infty)$.

Now we derive from the steal bound a bound on the total number of task creations in the Weld execution. Each stolen task can have at most $O(T_\infty)$ descendants that are not stolen themselves (and have no more recent stolen ancestor than this one). Consider the execution of a stolen task. The executing
worker’s queue will initially be empty, and it must create a task to put in the queue. This task will be no larger than half the size of the stolen parent task (with “half the size” defined in the same manner as for the equivalent Cilk execution above). If this newly created task is stolen itself, it will not count toward the number of descendants of the stolen parent task that were not stolen themselves, and the parent task will be at most half of its original size. If the newly created task is not stolen itself before the halved parent task is completed, it will be removed from the queue by the worker and executed. So in both cases when the queue is empty again, there is at most one additional descendant task of the original stolen task that was not stolen itself, and the currently executing task (which is either a reduced version of the original parent task or an unstolen descendant of it) is at most half the size of the task that was executing when the queue was last empty. Thus, the maximum number of unstolen descendant tasks of a stolen task is the maximum path length from the program root task to a single-iteration innermost loop task in the \texttt{cilk\_for} program, which is $O(T_\infty)$. Since every executed task is either a stolen task or an unstolen descendent of a stolen task, we arrive at our bound of $O(PT_\infty^2)$ on the total tasks created in a Weld execution.

Since $T_\infty$ is a sum of logarithmic terms, even $T_\infty^2$ should not be too large. As long as the number of task creations is about 10000 times smaller than the total number of innermost loop iterations executed, task creation overhead tends to be negligible. This is the case on a number of benchmarks, as our evaluation shows – Weld is competitive with handwritten OpenMP [29] code that creates no tasks and statically partitions work across threads.

Thus, even with an effective grain size of one (and thus full parallelism), our runtime system enjoys low task creation overhead. In contrast, using a grain size of one on all \texttt{cilk\_for} loops in an equivalent \texttt{cilk\_for} program would result in a created task for every innermost loop iteration. In order to expose enough parallelism to ensure good load balancing while still limiting task creation overhead, grain sizes must be carefully tuned in Cilk, and it is hard to develop heuristics for selecting these grain sizes in an automatic code generation system like ours.
Chapter 6

Evaluation

We evaluate Weld using several applications and ports of existing data processing systems. In our evaluation, we seek to answer the following questions:

1. Can Weld’s generated single-threaded and multicore code match the performance of existing specialized high performance systems? (§ 6.1)

2. How much does Weld speed up current data processing frameworks? (§6.2)

Experimental Setup. We run most benchmarks on a machine with an Intel Xeon E5-2680 v3 CPU with 12 cores (24 HyperThreads), based on the Haswell micro-architecture. Our handwritten baselines are compiled using Clang 3.5, with optimizations enabled (-O3, -march=native, -lto). We use HyPer [27] v0.5 for comparing TPC-H queries, and Cilk [4] for comparative experiments on parallel work-stealing. Each result is an average of 5 runs.

6.1 CPU Micro-benchmarks

To evaluate whether Weld’s CPU code generation meets the performance of hand-optimized baselines, we express several standard data processing workloads in Weld’s IL. We chose a selection of TPC-H queries, PageRank graph processing algorithm, and a logistic regression classifier. For each, we compare the performance of the Weld program to a hand-optimized baseline implementation and an existing high-performance implementation. Our hand-optimized implementations are written in C using Intel’s AVX2 intrinsics library, and optimization techniques such as predication; they represent our best efforts to represent the achievable performance on the hardware. The results show that the automatically optimized Weld
TPC-H queries. Figure 6-1 shows Weld’s execution time for four TPC-H queries executed at scale factor of 10, compared to the HyPer database and our handwritten implementations, which used OpenMP [29] for multithreaded execution. Weld matches or outperforms HyPer by up to $5 \times$ (Q6) and compares competitively with our handwritten baseline implementations for every query, validating that Weld’s code generator is competitive with the best known code generating database to date.

PageRank. Figure 6-2a shows the results for Weld’s performance on the PageRank algorithm on the widely used twitter_rv graph, comparing against a well-tuned handwritten implementation using Cilk [4] for load balancing and against GraphMat [34], which has the fastest multicore PageRank implementation we found. Weld’s per-iteration runtime for both serial and parallel code is competitive with Cilk, and outperforms GraphMat.

Logistic Regression. We also implemented a logistic regression classifier in Weld and compare against a single-threaded off-the-shelf implementation of logistic regression using Eigen [18]. We ran the classifier on a linearly separable dataset with 100k points and 10 features, and ran 100 iterations. Figure 6-2b shows the results; Weld’s runtime was competitive with the hand-tuned Eigen implementation.
Figure 6-2: *Microbenchmarks*. 6-2a. Weld’s CPU backend outperforms GraphMat [34] and matches our handwritten C on the twitter_rv graph. 6-2b. Weld single-threaded logistic regression performance is competitive to that of Eigen.

Figure 6-3: *Library Integration*. 6-3a. Weld outperforms TensorFlow (TF) by 4.7× on one thread and 5× on 12 threads for nearest-neighbor classification. 6-3b. Weld on a single core improves Word2Vec performance by 16× over TensorFlow’s standard operators. 6-3c. Replacing Java codegen with Weld in Spark SQL improves TPC-H Q1 by 3.8× and Q6 by 1.7× on a single core.

### 6.2 Existing Frameworks

We have integrated Weld with several high-level libraries for parallel data processing. We evaluate the performance benefits of Weld’s integration.

**TensorFlow.** *Nearest Neighbor Classification.* To evaluate the performance benefit of our TensorFlow integration, we first ran a nearest neighbor classifier acquired from a set of sample TensorFlow applications [28]. The classifier predicts the label of a test example as being the label of its nearest neighbor
(based on L1 distance) in a training set. We use a training set of 55,000 images from the MNIST data set [14] and record average prediction accuracy over 500 test images.

Figure 6-3a compares Weld’s single and multi-threaded performance to vanilla TensorFlow and a handwritten baseline which materializes no intermediate data and uses vector instructions. Weld’s single-threaded execution is competitive to the optimized baseline (a difference of 36%), and outperforms TensorFlow by 4.7$\times$ by fusing operators and preventing intermediates while also vectorizing arithmetic. The parallel Weld execution matches the baseline and substantially outperforms TensorFlow.

**Word2Vec.** We also evaluate TensorFlow on a realistic workload called Word2Vec [38], which maps words in a vocabulary to a smaller $D$-dimensional space so semantically similar words (like ‘cat’ and ‘dog’) appear close together. The task is common in natural language processing pipelines. TensorFlow provides two methods for computing Word2Vec: one using standard TensorFlow ops using the Python API and one using custom handwritten C++ op provided by TensorFlow.

Figure 6-3b shows the results; compared to TensorFlow’s custom op, single-threaded Weld performs roughly 2$\times$ slower but improves performance by 16.5$\times$ over the standard TensorFlow API; Weld’s transforms greatly reduce the number of intermediates results materialized and also allows for techniques such as vectorization across TensorFlow ops.

**Spark SQL.** To evaluate the SparkSQL code generation, we implemented TPC-H Queries 1 and 6 and the compared the execution time (i.e., excluding data loading time) of the generated Weld code to the generated Java code on the TPC-H dataset (scale factor = 10). We compare serial performance because each of Spark’s tasks runs on a single thread, and its engine automatically runs different tasks on different cores across a cluster.

Figure 6-3c shows the results; compared to the generated Java code, Weld improves performance by 3.8$\times$ for Q1 and 1.7$\times$ for Q6. Because Spark SQL performs optimizations such as loop fusion manually using its own optimizer, much of this difference comes from avoiding Java-specific overheads. This experiment illustrates that even when higher level frameworks run their own domain-specific optimization passes, noteworthy performance improvements are possible by generating machine code.
Chapter 7

Related Work

A number of prior systems look at executing parallel applications efficiently. OpenCL [32] and SPIR [20] provide an interface to diverse parallel hardware. They let users launch multiple copies of a kernel function in parallel, but use a sequential representation (C- or LLVM-like code) for the function. Output from the kernel functions is through shared memory. In contrast, Weld supports nested parallel loops that are amenable to complex optimizations such as loop blocking, and has explicit data types (builders) for constructing parallel results, which can be implemented differently on different hardware.

Dandelion [31] compiles .NET LINQ expressions to heterogeneous hardware. It uses LINQ (relational) operators as its code representation, which makes it higher-level but also more restricted than Weld. For example, Dandelion does not support nested parallel loops, which makes it difficult to express optimizations such as loop blocking, and it provides group-by and join operators directly rather than giving lower-level primitives like the dictionaries and builders in Weld.

NESL [3], Data-Parallel Haskell [10] and other parallel functional languages use functional operators as their IR. While they support some types of fusion transformations, this representation makes it difficult to express other general transformations that can be captured in loops and builders, such as loops that produce multiple results (§3.3) and loop blocking.

There is a long history of DSLs for a variety of purposes, including improving optimizability and parallelization [12]. In Weld, we focus on developing a language that is both optimizable and expressive enough to capture many data-parallel applications, while also supporting cross-library optimization. Delite [8] is a compiler framework for Domain Specific Languages (DSLs) that can map to diverse...
hardware. Its latest release uses an IL called DMLL [7] with constructs called *multiloops* and *generators*, which resemble Weld’s parallel loops and builders. DMLL is more restricted, however: each multiloop can only produce at most one output per generator on each iteration, and the results of generators are implicitly materialized after the loop finishes; in contrast, Weld can merge multiple results into each builder on each iteration (*e.g.*, by passing it to a nested loop), and can have multiple loops that update the same builder (*e.g.*, to implement list concatenation). Delite also operates primarily at compile time [33], whereas Weld can optimize across libraries that are dynamically combined at runtime (*e.g.*, in a Python interpreter session).

Builders are similar to Cilk reducers [13] and to LVars [21]. Unlike reducers, builders cannot have races [24]. Unlike LVars, builders cannot be “read” until all writes have finished, which makes them more restricted but also simple to implement.

Finally, runtime code generation is used for performance in more scoped domains, such as SQL engines [27, 2, 9]. These systems are often complex to write because they need to generate imperative code directly from multiple operators (*e.g.*, fusing them) [27]. Weld makes code generation simpler because it automatically performs optimizations across operators.
Chapter 8

Conclusion

We have presented Weld, an intermediate language and runtime that can run data-parallel applications efficiently on modern hardware through two key contributions. First, Weld offers a restricted intermediate language, based on nested loops and parallel builders, that can express a wide range of data-parallel workloads but is easy to optimize. Second, Weld’s runtime API performs complex optimizations across library functions used in the same program, enabling speedups that are not possible with today’s low-level ILs and separately written libraries. We have shown both analytically and empirically that the multicore CPU backend for Weld performs well.
Bibliography


