Contech: Parallel Program Representation and High Performance Instrumentation

Brian P. Railing
Georgia Institute of Technology
brian.railing@gatech.edu

Abstract
This summary of my dissertation work explores a pair of problems: how can a parallel program’s execution be comprehensively represented? How would this representation be efficiently generated from the program’s execution?

I demonstrated that the behavior and structure of a shared-memory parallel program can be characterized by a task graph that encodes the instructions, memory accesses, and dependencies of each component of parallel work. The task graph representation can encode the actions of any threading library and is agnostic to the target architecture. Subsequently, I developed an open source, LLVM-based instrumentation framework, Contech, for generating dynamic task graphs from arbitrary parallel programs. The Contech framework supports a variety of languages, parallelization libraries, and architectures, with an average instrumentation overhead of less than 3x. Various analyses have been applied to Contech task graphs, including modeling a parallel, reconfigurable architecture.

1. Introduction
Modern hardware is going through a paradigm shift. The solutions and techniques of the past decades are reaching physical limits. Already architectures are embracing parallelism to continue to provide improved performance. Many opportunities are looming ahead, such as 3D stacking, heterogeneous components, and dark silicon. Designing for these opportunities requires greater understanding of the software.

Software is ever increasing in size and complexity. To continue its development and improvement, programmers must understand their software. How does the software work with the compiler, hardware, operating system, and other programming libraries? To understand the software and how it is executing, tools and analyses are required. Without quality tools and analyses, software will not be able to fully utilize the resources provided, especially as the computing ecosystem continues to evolve and change.

Therefore, there is a continuing need of programmers, compilers and computer architects to understand modern programs, and each user in different ways [4]. The common approach has been to develop a targeted instrumentation and corresponding analysis for each particular need. In many circumstances, existing instrumentation frameworks have been available to assist in collecting the specific data about the program, mitigating some of the work to develop the instrumentation. While some instrumentation is designed to have negligible overhead (e.g., using sampling), allowing these tools to be used with production systems, there is still a need for more detailed insight into parallel programs.

This paper details three contributions I have made to the fields of compilers and program analysis:
• A novel task graph representation of parallel programs that supports independence of architecture, language, and threading paradigm. This representation splits synchronization tasks from work tasks, and includes the memory and execution trace.
• The design and implementation of a framework in the popular LLVM compiler infrastructure to efficiently generate the dynamic Contech task graph representation for arbitrary parallel programs.
• Demonstration of the comprehensive nature of the task graph representation by modeling the parallel programs on a parallel, reconfigurable architecture.

The next three sections of this summary will cover each of the contributions in turn. Section 2 will briefly describe the task graph representation developed and used by Contech. Section 3 will describe Contech’s instrumentation and its performance. Section 4 covers one example of modeling parallel programs using Contech task graphs. And Section 5 concludes this dissertation summary.

2. Task Graph Representation
(This work previously appeared in TACO [19].)
3. Contech Framework

(This work previously appeared in TACO [19].)

The second, and more pronounced, contribution of this dissertation work is to generalize the instrumentation for the collection of task graphs from parallel programs, and to collect the graphs efficiently. Prior work has relied on custom runtime libraries or annotations that are added to the program in order to identify the parallel program’s task graph. Similar task graphs were generated for HPF-based programs [1]; however, the work was never generalized. Instead of modifying the libraries or annotating every program, I identified commonalities across parallel API sets and extended a compiler to, in effect, annotate the programs appropriately.

Using LLVM, I developed instrumentation for a diversity of parallel programs. Contech’s instrumentation is currently limited to parallel programs meeting the following constraints. First, the program uses a combination of pthreads, OpenMP, MPI, or Cilk to implement its parallelism. Second, the program is written in C, C++, or Fortran. And third, the program is compiled for the x86 or ARM ISAs.

3.1 Generating a Task Graph

Contech’s task graph generation begins with two components: a compiler pass for LLVM [15] and a runtime library linked to every application. The results herein used LLVM 3.4 with OpenMP support. Contech relies on Clang (the C / C++ LLVM front end) or Dragonegg (a plugin for gcc to support Fortran and other languages) to generate the intermediate representation (IR) for the program. The compiler pass modifies the IR to include the instrumentation for recording the actions of the parallel program. The pass operates function-by-function and identifies whether the function should be instrumented, or treated as a black box (e.g., malloc, pthread_create, etc). Black box functions are from a defined list; and although the compiler can detect certain operations such as atomic accesses, the programmer may have user-defined versions that would need to be explicitly identified to the instrumentation pass. This way each function only has one static representation in the task graph: either it is a sequence of basic blocks or it is an abstracted functionality such as allocating memory or creating a thread.

The design of the instrumentation is for each action of interest (computation graph actions) by the program to be recorded by a call to an instrumentation routine in the runtime library. To be architecture-independent, the instrumentation is written in C, with functions for each type of event. Inserting multiple function calls per basic block would have a severe performance impact, so Contech requires programs to be compiled with Clang’s -f1to flag. This flag instructs clang to perform link-time optimization, inlining the instrumentation routines. The most common instrumentation call, recording a basic block event, requires 4 inlined x86 instructions, which are off the critical path.

The generated event list trace is then converted into a Contech task graph. To avoid any disruption of the running program, this processing does not take place until the instrumented program has completed.

3.2 Performance of Contech Instrumentation

This section presents performance measurements from running the PARSEC and NAS benchmark suites on the clus-
ter of machines configured per Table 1. Figure 1 shows the performance of two Contech implementations, along with a comparable Pin tool implementation (see Section 3.3). The first Contech performance measurement was published in TACO [19], while the Contech Current includes further optimizations developed since that publication to achieve a final slowdown between 1x to 5.5x and averaging 3.0x.

3.3 Comparison with Other Tools
I will highlight four common frameworks that provide APIs to record sufficient detail to generate a task graph equivalent to Contech’s instrumentation. Pin [17] is a well known framework designed for highly customized data collection, typically toward analyzing one aspect of a program. Valgrind [18] is a heavy-weight instrumentation platform that focuses on the usage and access to memory in a program. DynamoRIO [8] provides an API toward instrumenting and optimizing the instructions in a program. PEBIL [16] uses Dyninst as part of sampling programs’ memory accesses, recording a 2.9x slowdown at a 10% sampling rate.

Pin has support for collecting similar data to Contech’s task graphs. Based on several tools previously published [5] [13] [16] [23], we developed an optimized Pin plugin that was comparable to Contech’s data collection. Running the plugin on the same benchmarks as Contech, the slowdown from the Pin instrumentation is also shown in Figure 1. The average slowdown from Pin was 15.8x, significantly more than the 3.0x from Contech.

4. Architectural Modeling
(The following results previously appeared in PACT [21].)

The following contribution demonstrates the value of the task graph representation by modeling and predicting parallel program performance using every component of the task graph representation. Previously, hand-generated task graphs were analyzed to predict parallel program performance [2]. Here a possible reconfigurable system will be explored. In this system, the architecture provides the capability to reconfigure based on the explicit directive of the program, enabling the hardware to match the requirements of the program. The compiler will analyze the program and its source to determine the appropriate configuration for each section of the program. Particularly by using Contech’s task graph representation, I will analyze parallel programs at the task granularity and show that even at this coarse grained unit of execution, programs differentiate and specialize their execution to specific resources.

To analyze a program, I extended the architectural model from Cabezas and Püschel [9] to use Contech’s task graph representation as input. This model takes in a program as a sequence of LLVM IR operations, which has similarities to assembly. The model only handles single-threaded execution, so two other components were developed. First, a task graph analysis measured the communication and data sharing between contexts. This metric can then inform the architectural model to mimic the effects of data movement in a single-threaded simulation. Second, an additional analysis took the estimated speedups produced by the architectural model and applies each to its corresponding task in the task graph. By respecting the order and separating out synchronization and other operations, the rate-based improvement represents an improvement in that task’s execution time and the overall program order then reveals whether this improvement impacts execution time.

Starting from an Intel-based configuration, experimentally we found that the dynamic addition of a single function unit tied to the requirements of the task can improve the execution of parallel programs by 1.36x. Furthermore, the analysis determined that specifying the requirements statically based on this profile would achieve better than 90% of the speedup available to the dynamic approach. Work is ongoing to quantify the costs and savings from this technique.

5. Conclusion
This dissertation work puts forward a single, common description of parallel programs demonstrated by generating the representation from parallel programs across a diversity of languages, architectures, and parallel runtimes. The Contech framework collects the task graph with less overhead than any comparable approach to minimize the program’s
perturbation. And the usability of the representation has been explored through a variety of analyses, including modeling a parallel reconfigurable architecture. This work extends the field by providing a framework for the efficient instrumentation and detailed analysis of parallel programs.

References


