Contech: Efficiently Generating Dynamic Task Graphs for Arbitrary Parallel Programs

BRIAN P. RAILING, ERIC R. HEIN, and THOMAS M. CONTE, Georgia Institute of Technology

Parallel programs can be characterized by task graphs encoding instructions, memory accesses, and the parallel work's dependencies, while representing any threading library and architecture. This article presents Contech, a high performance framework for generating dynamic task graphs from arbitrary parallel programs, and a novel representation enabling programmers and compiler optimizations to understand and exploit program aspects. The Contech framework supports a variety of languages (including C, C++, and Fortran), parallelization libraries, and ISAs (including ×86 and ARM). Running natively for collection speed and minimizing program perturbation, the instrumentation shows 4× improvement over a Pin-based implementation on PARSEC and NAS benchmarks.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Tracing*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Parallelism and concurrency*

General Terms: Performance

Additional Key Words and Phrases: Instrumentation, parallel program modeling, task graph

ACM Reference Format:
Brian P. Railing, Eric R. Hein, and Thomas M. Conte. 2015. Contech: Efficiently generating dynamic task graphs for arbitrary parallel programs. ACM Trans. Architec. Code Optim. 12, 2, Article 25 (July 2015), 24 pages.
DOI: http://dx.doi.org/10.1145/2776893

1. INTRODUCTION

The performance of emerging multicore processors can only be fully utilized by optimized, well-designed parallel programs. Achieving this goal has been the focus of significant research. To analyze and understand the diversity of parallel programs, a single common representation is needed. The task graph can provide this representation and is well established in the study of efficient scheduling of parallel tasks [Kumar et al. 2007; Sridharan et al. 2014; Vandierendonck et al. 2013; Yoo et al. 2013], as well as evaluating future architectures [Almeida et al. 1992; Etsion et al. 2010] and parallelizing applications [Gupta and Sohi 2011].

Current state-of-the-art task graph generation is not sufficient to support this usage. Most approaches are only collecting a task graph to immediately schedule the program, and others are targeted at one specific application such as measuring a program's

This work was supported in part by NSF Grant #1217434.
Authors' addresses: B. P. Railing, School of Computer Science; email: brian.railing@gatech.edu; E. R. Hein, School of Electrical and Computer Engineering; email: ehein6@gatech.edu; T. M. Conte, School of Computer Science Joint with School of Electrical and Computer Engineering; Georgia Institute of Technology 266 Ferst Dr. Atlanta, GA 30332; email: conte@cc.gatech.edu.
Contact author: B. P. Railing, email: brian.railing@gatech.edu.

DOI: http://dx.doi.org/10.1145/2776893

parallelism. The identification of a task graph can require programmers to add additional annotations to their programs, beyond the ones required to make it parallel. Other work restricts the scope of programs to those from a specific language or threading model. Furthermore, if a tool does generate a task graph or other representation, it is still hampered by significant slowdown.

This work presents Contech, an open source, compiler-based framework [Railing and Hein 2015] that addresses these problems by collecting a trace of the program's execution and generating a task graph from that trace. The framework requires no additional annotations by the programmer, only using what is already present in the code. Nor are programmers restricted to one language, as the tool instead relies on LLVM Intermediate Representation (IR). Furthermore, by targeting the IR, the Contech framework can span a variety of instruction set architectures (ISAs), as well as avoid capturing architecture-specific details, such as calling convention and register spills. Contech can currently generate task graphs from arbitrary parallel programs across the following combinations of language (C, C++, or Fortran), parallelization library (pthreads, OpenMP, or MPI), and ISA ($\times$86 or ARM).

By converting the parallelism operations to a common set of synchronization types (Create, Join, Sync, Barrier), Contech's task graph can represent any program written in a shared- or distributed-memory threading model, be it task-based, fork-join, etc. Additionally, the Contech task graph includes data accesses as well as a basic block execution trace, which extends the possible applications of the representation. Finally, we have focused on having an efficient design of the instrumentation in order to reduce the perturbation to parallel programs. As a result, it is significantly faster than all but the most limited of competing instrumentation approaches.

In rising above any one combination of language, parallelization library, and ISA, architects, compiler writers, and programmers are able to use Contech to explore a diversity of programs without being tied to the specific details of the underlying system. Therefore, "by abstracting away architecture-specific details, many [analyses] can work across the [different] architectures with little porting effort" [Luk et al. 2005]. Contech task graphs are a novel, rich representation of a program's execution. We believe that unifying a memory trace with path information and cross-thread dependencies will enable deeper analyses than can be carried out on either an instruction or memory trace in isolation.

This article details four contributions we make 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 (parallelism shaping) tasks from work tasks, and includes the memory trace and execution information.
—The design and implementation of a framework in the popular LLVM compiler infrastructure to efficiently generate the dynamic Contech task graph representation for programs with arbitrary parallelism within the supported frameworks.
—The performance features in Contech's task graph collection framework that impose only 3$\times$–5$\times$ slowdown from instrumentation, which is four times faster than a similar implementation using Pin.
—Three sample analyses using Contech's task graph representation to explore the level of parallelism and other characteristics of parallel programs.

The rest of the article is as follows. Section 2 discusses related work. Section 3 introduces our innovative task graph representation. Section 4 discusses the architecture and workflow of the Contech tool. Section 5 presents three example analyses of Contech task graphs. Section 6 presents the performance features in Contech's instrumentation

and compares the front end with a Pin-based approach. Finally, Section 7 concludes the work.

2. RELATED WORK

Contech extends prior task graph representations by including the memory trace and execution information, as well as labeling partitions of the graph based on how each task affects the available parallelism of the program. The task graph representation was first used to analyze program scheduling by decomposing the program into tasks [Blelloch et al. 1997; Blumofe and Leiserson 1993; Gerasoulis et al. 1990]. Additional research extended task graphs with their potential interactions [Long and Clarke 1989] and found subsets of the graph with no interaction and representing the subset as a single node [Beckmann and Polychronopoulos 1992]. Parallel Program Graphs [Sarkar and Simons 1994] is an extension of control flow graphs, which adds a create parallelism node type (*MGOTO*) and synchronization edges; however, other types of tasks affecting parallelism must be inferred by the edges present in the graph. In Structured Parallel Programming [McCool et al. 2012], the authors added fork and join symbols to their graphical notation.

Contech is not the first work to propose task graphs and generate them from instrumented programs. However, prior approaches have been limited to specific languages and program structures, or required that programmers add additional annotations to their code so that a tool could construct a task graph. Nabbit [Agrawal et al. 2010] and Vandierendonck et al. [2013] both extend Cilk with annotations to identify tasks for scheduling. Even hardware-based approaches have also relied on programmer annotations made to existing parallel C or C++ programs [Etsion et al. 2010; Gupta and Sohi 2011; Kumar et al. 2007]. One approach, Varuna [Sridharan et al. 2014], identifies virtual tasks out of parallel programs, using existing parallelism Application Programming Interfaces (APIs) to identify when parallel work is created, and dynamically finds efficient schedules of the running programs.

Adve and Sakellariou [2001] demonstrated a framework that can collect task graphs without any programmer annotations, when the program is written in High Performance Fortran (HPF). Yet, when they extended their approach to more general programs, they instead relied on hand-generated task graphs, as they noted, "[we] manually wrote scripts to generate each of the task graphs" [Adve and Vernon 2004]. Thus, at the time, no compiler-based approach was sufficiently powerful to collect the necessary task graphs. We propose that Contech is such an approach.

Collecting a task graph for an arbitrary program requires a generalized instrumentation framework. While we elected to build our own instrumentation (as discussed in Section 4), there are many existing frameworks that could be used to record an appropriate trace that would become a task graph. These frameworks are generally designed to instrument binaries. Pin [Luk et al. 2005] is a well-known framework designed for highly customized data collection, typically toward analyzing one aspect of a program. The original Pin included support for ARM ISA; however, this support has not been maintained. Rico et al. [2011] used a combination of Pin and a runtime library to capture a memory and instruction trace along with identifying the parallel operations for use with a simulator. Valgrind [Nethercote and Seward 2007] is a heavyweight instrumentation platform that focuses on the usage and access to memory in a program. DynamoRIO [Bruening et al. 2003] provides an API toward instrumenting and optimizing the instructions in a program. PiPA [Zhao et al. 2008] uses DynamoRIO to collect a control-flow and memory operation trace; however, it requires additional software threads to process the trace. To quote the authors, "PiPA essentially obtained good performance at the expense of additional resources" [Zhao et al. 2010]. Additionally, PiPA has limited support for parallel threads and no support for any operations

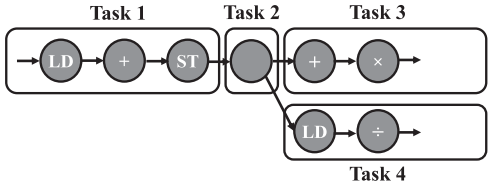

Fig. 1. Aggregating a computation graph into tasks (Work: 1, 3, and 4; Create: 2).

between threads (syncs, forks, joins, etc.). PEBIL [Laurenzano et al. 2012], with a similar functionality as other frameworks, also supports sampling the added instrumentation in order to lower overhead. DBILL [Lyu et al. 2014] uses QEMU to generate architecture independent instruction sequences that are translated to LLVM IR for simple binary instrumentation. We also note that while our trace format achieves our performance goals, prior work has proposed several extensions to improving the trace representation [Goel et al. 2003; Larus 1999; Tallam and Gupta 2007].

Contech is built on the LLVM compiler framework [Lattner and Adve 2004], which provides the ability to observe and manipulate the intermediate representation of a program. The Contech implementation significantly extends the features from several other prior works, such as Harmony [Kambadur et al. 2012], which relies on per-thread instrumentation added by LLVM to gather basic block execution counts across a parallel program and keeps track of the thread count for each block. Peregrine [Cui et al. 2011] uses LLVM to record a synchronization trace for creating a deterministic schedule of the program. Clikview [He et al. 2010] records aspects of a Cilk program's execution to construct a DAG and estimate the program's parallelism. Contech's task graph representation is more general than that collected by any of the prior work, as no instrumentation has practically combined synchronization and scheduling information with the control-flow and data accesses of the program.

3. TASK GRAPH REPRESENTATION

3.1. Parallel Program Representation

A parallel program can be represented as a set of tasks that may execute in parallel and interact via both actions and data dependencies. A task graph models the behavior and execution of a parallel program as a directed acyclic graph, where nodes are tasks and edges are the explicit dependencies between tasks.

$$G = (V, \overrightarrow{E}), \forall v \in V :< task >$$
$$\forall \overrightarrow{e} \in \overrightarrow{E} :< scheduling\ dependencies >.$$

While this article focuses on shared-memory programs, a distributed-memory program can be similarly represented where all memory accesses are mapped to disjoint sets of addresses, and there exist send and receive actions that allow accesses between these sets [Gerasoulis et al. 1990].

To derive a task graph, we first begin with a program's execution, which is considered as a computation graph. In the computation graph, every node is an action, be it an instruction, basic block, function, or some other unit of execution; and Contech supports actions that are either basic blocks or functions. Some actions have additional explicit nondata dependencies on prior actions, and are termed *synchronization actions*. Programs may use these dependencies to enforce an order on its data dependencies. The actions are then aggregated together into tasks, thereby transforming a computation graph into a dynamic task graph. Figure 1 shows an aggregation of compute nodes into tasks.

We define a task as containing either a sequence of actions taken without an intervening synchronization action, or the synchronization action itself. We identify tasks consisting of the former as work tasks (task partition W), which may contain individual instructions, basic blocks, or entire functions. In practice, most of our tasks are sequences of basic blocks; however, a standard system function such as `malloc` is stored intact in a task, although other works have used different sequences [Blelloch et al. 1997; Blumofe and Leiserson 1993]. Tasks may also contain a set of data locations accessed.

3.2. Contech Task Graph

A unique feature of Contech task graphs is that tasks containing synchronizing actions are split into four partitions on the task graph: creates (C), joins (J), syncs (S), and barriers (B), which along with the fifth partition, work (W), cover all tasks in the graph. Each *synchronization action* can only map to a single partition type in the task graph.

C, J, S, B, W are partitions on V:$< task\_type >$.

These synchronization tasks define the topology of the task graph, and cleanly separate the synchronization from the useful work within the graph. Using these partitions, the task graph has one type of edge, and the cause of dependency edges is clear from the partitions of the tasks, whereas prior task graph representations either stored the cause as an action in the tasks or as different edge types. Thus, the dependencies in the task graph are only those actions explicitly invoked by the program to order otherwise concurrently executing contexts. In practice, the particular mapping of parallelism functions to the partitions is left to tools that generate task graphs.

For the purposes of this work, we use the term *parallelism* to refer to observed simultaneous execution and *potential parallelism* to mean the exposed possibility for two (or more) things to execute at the same time. The following properties are expected of each partition regardless of the architectural details of the generating tool. Create tasks increase the potential parallelism in the task graph, commonly having an *out* degree greater than their *in* degree. Join tasks are the complement of creates, in that they reduce the potential parallelism in the task graph. Sync tasks capture actions that impose an order between tasks, but without changing the parallelism in the task graph. The sync task may encapsulate a semaphore or condition variable, where one task is waiting on another task's notification. Tasks in the sync partition can also represent lock acquires and releases, or other atomic operations. The particular mapping is again left to tools implementing the task graph definition and could capture other functionality. Barrier tasks are similar to sync tasks, except that the ordering requirement is all to all rather than one to one (e.g., lock) or one to many (e.g., condition variable). Together, the dependencies in a task graph establish a partial order between the tasks. While it may be possible to create a total order using additional annotations such as the start and end times of tasks, such an order would be an artifact of a particular execution.

The set of work tasks can also be partitioned into disjoint subsets, where each subset represents a *context*. This partitioning of work tasks can be constructed using information known for each execution action. The practical definition of a context depends on threading models, but in general such a subset in the task graph represents the implicit data dependencies that many architectures carry between work tasks via registers, stack locations, return values, etc. As such, tasks in a context are expected to depend on prior tasks in the context and may not be able to execute in parallel with each other, even if there are no explicit data dependencies. Nonwork tasks are also

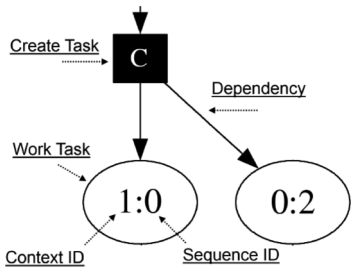

Fig. 2. Legend for visualized Contech task graphs.

partitioned into specific contexts, associated with the work executed prior to and following the nonwork action.

Contech work tasks contain sequences of execution and memory actions, which are in the program order and may differ from the order dictated by the microarchitecture executing the program. This property of the task graph also holds for any data locations accessed by the task. Dependencies between these actions may restrict the set of possible reorderings that the compiler or hardware can exploit; however, as the task graph is independent of the architecture, it does not preserve the actual execution order of individual actions within a task (such as memory accesses), only the program order. The task graph contains edges that establish a partial order between work tasks. This ordering of tasks is defined and restricted by the program to ensure correctness, in part by preserving the intended dependencies between memory writes and reads in different threads.

The Contech task graph also contains properties and annotations such as a set of predecessor and successor tasks for each node, as well as start and end times. Being annotations, the start and end times are not used in constructing the task graph, and are only provided for certain analyses.

3.3. Visualizing a Contech Task Graph

It is useful to visualize the elements of Contech task graphs. Figure 2 provides the legend for the subsequent figure. Tasks are uniquely labeled with a Task ID, which is the pair of <Context ID>:<Sequence ID>. The Context ID is a unique identifier for a context, akin to a thread ID. The Sequence ID is a monotonically increasing identifier in each context. Work tasks are represented by ovals, with their Task ID visible. Other task types have a Task ID, but are represented instead by their type. The edges in the graph are represented as arrows, with the direction of the arrow pointing toward the dependent task. Synchronization actions are black squares labeled with a single letter indicating the type. Note that while the implementation also assigns identifiers to these actions, they are not shown in the visualization. While these images are taken from actual task graphs, in our experience most parallel programs use thousands to millions of tasks during execution, which exceeds the capabilities of even a simple graphical visualization.

Figure 3 shows each of the task types from a task graph. The Create task (Figure 3(a)) shows one task entering the create task and two tasks dependent on the create. The 1:0 shows that Context 1 has been created, and therefore Context 0 invoked the create. Join (Figure 3(b)) has two tasks entering the join task, and one task (Context 0) continuing. Syncs (Figure 3(c)) involve two tasks entering and leaving the sync, which represent the two chains of dependencies. In one chain, each sync (e.g., lock acquire/release) must occur after its opposite; for example, a lock is acquired after being released. The second chain follows different contexts as each operates on the sync construct (e.g., lock). The

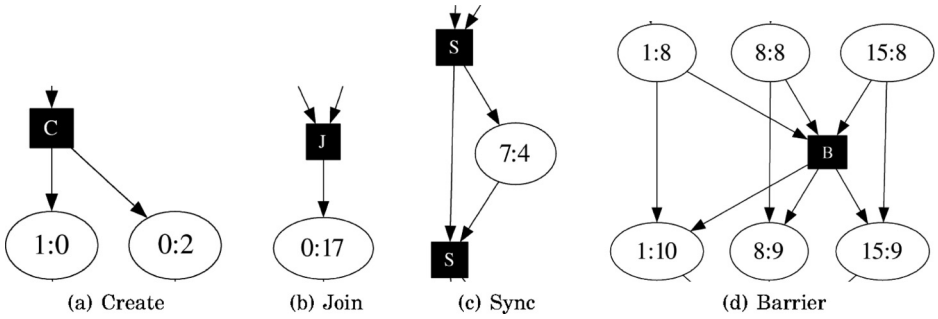

Fig. 3. Contech task graph features by type (nonwork nodes also have task IDs; for example, the Barrier node (B) is 1:9).

final task graph feature, the barrier (Figure 3(d)), involves many tasks entering and leaving. Tasks after the barrier are dependent on both their context's last task as well as the barrier task itself, although the former can be concluded transitively from the latter.

3.4. Incorporating a Parallel Programming Model into the Task Graph Representation

The process of integrating a new parallel programming model is straightforward. Each construct within a parallel programming model must be mapped to the appropriate representation in a task graph. The construct is considered for how it affects the (potential) parallelism of the program: is it increased (create), decreased (join), or has the parallelism remained the same (sync / barrier). The intent of the programmer also influences classification; for instance, atomic instructions can be used for fine-grained concurrent access to shared locations and should therefore be represented explicitly as sync tasks rather than be contained within work tasks.

When a representation for a construct is established, the construct must be appropriately instrumented to record the necessary information to represent the task. For create and join tasks, this often requires identifying the contexts involved in the operation. Sync and barrier tasks are identified by the address of the dynamic instance of the construct. This instrumentation is at the core of the Contech framework.

Condition variables, part of the POSIX threads standard, provide a way for threads to signal both 1:1 and 1:many. The core design is centered on representing condition wait, which performs three operations: it unlocks the mutex, after some time the condition variable is signaled, and the mutex is reacquired. Each of the three operations is represented as a separate sync task performed on either the mutex or condition variable, with the mutex's or the condition variable's identifier (e.g., address) stored in their respective tasks.

Figure 4 shows how a simple OpenMP program maps to a Contech Task Graph. From top to bottom, the parallel region creates two parallel contexts, which then execute up to the *single* directive. The contexts synchronize and one enters the region, while the other skips to the end of the region. The end of the *single* construct implies a barrier within this parallel region. The contexts continue executing, until they reach the end of the parallel region, where they wait at the implicit barrier and then join back into the serial execution.

OpenMP 3.0 introduced the *task* keyword, which was extended in 4.0 to add a *depend* keyword. As the OpenMP task increases the potential parallelism, it is first bracketed by Contech create and join tasks. Then each OpenMP task dependency provides an ordering constraint between separate tasks without changing the program's exposed parallelism, and each is therefore represented by a Contech sync task for the associated

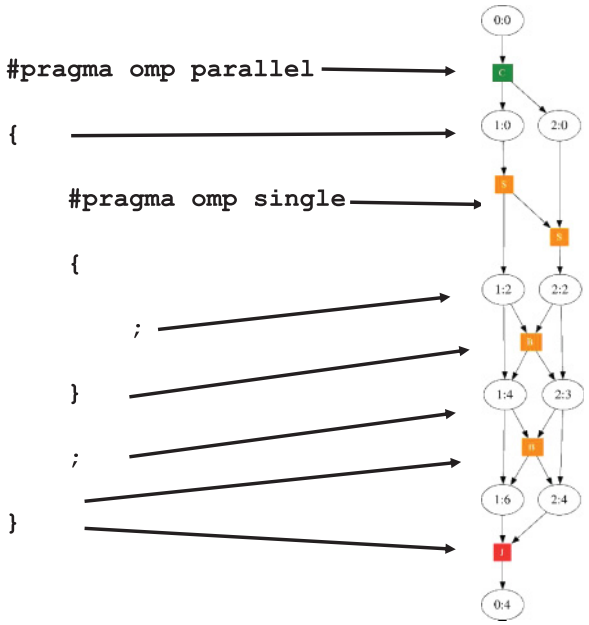

Fig. 4. Simple OpenMP program as a Contech task graph.

item. The constraint's identifier (e.g., address) is then used to link the dependencies together in the final task graph.

While the work in this section lays the groundwork for representing arbitrary parallel programs as Contech task graphs, Contech's instrumentation (described in Section 4.1) currently has three constraints on the "arbitrariness" of the parallel program. First, the program uses a combination of pthreads, OpenMP, or MPI to implement its parallelism. Second, the program is written in C, C++, or Fortran. And third, the program is compiled for the $\times 86$ or ARM ISAs. Ongoing work is exploring how to relax these constraints to successfully instrument and represent other parallel programs.

4. THE ARCHITECTURE OF CONTECH

The Contech framework is composed of two parts: the generation of a dynamic task graph and the analysis of task graphs. The front end and middle layer work together to generate a task graph from a benchmark's source code. Then back ends implement analyses and transformations that can be executed on this representation. The focus of this design is to allow easy analysis of the Contech task graph representation without a need for back end programmers to understand the details of how a task graph is generated. Each component will be explained in further detail in this section.

4.1. Front end

The first component is the Contech front end, which consists of two parts: a compiler pass for LLVM [Lattner and Adve 2004] and a runtime library linked to every application. The results in this article used LLVM 3.4 with OpenMP support, and Contech has been tested with LLVM 3.2 through 3.5. 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 IR for the program. The compiler pass modifies the IR to include the instrumentation for recording the actions of the parallel program. As Contech covers a variety of parallel libraries, the instrumentation is applied to the IR rather than

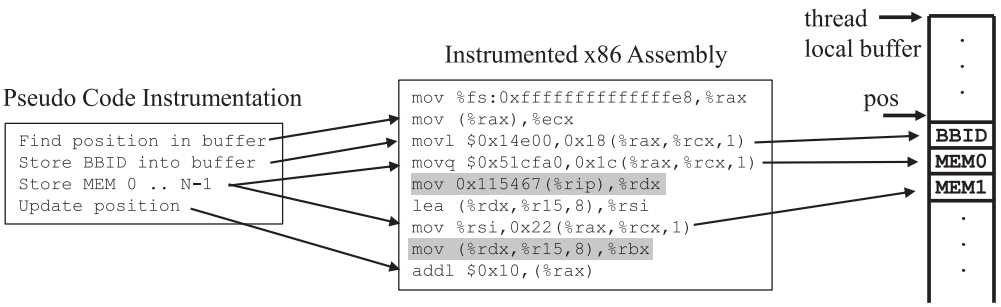

Fig. 5. Runtime instrumentation design (gray instructions are original code).

modifying the runtime libraries themselves. 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.

For each basic block, the LLVM pass finds instructions that will access memory and introduces instrumentation to record the properties of the memory access: address, size, and load versus store. Given that instrumentation occurs prior to register assignment, the LLVM-based instrumentation only records the memory operations specified by the program and not those imposed by a particular architectural model due to register allocation. Information about every basic block is also copied into a simplified set of debugging information, such that analysis tools can identify the function name, source file, etc., even when processing a basic block ID.

The instrumentation records each action of interest (computation graph actions, thus either IR instructions or function calls) by a call to an instrumentation routine in the runtime library that creates a corresponding event. The runtime instrumentation is written in C to be architecture independent, 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 `-flto` flag. This flag instructs Clang to perform link-time optimization, which inlines the instrumentation routines into the binary. The most common instrumentation call, recording a basic block event, requires four inlined $\times$86 instructions, after the optimizations discussed in Section 6.1. Figure 5 shows the steps required for recording a basic block event, as well as the optimized instrumentation inlined into the assembly code. In this example, the instrumentation stores the basic block ID, as well as updating the position in the thread-local buffer. The original code had two load instructions, so the instrumentation stores each address loaded by the original code. While the instrumentation introduces additional instructions, the original code has no dependencies on the instrumentation, thus maintaining the existing critical path.

Syncs are the most common complex event. Whether the sync is an atomic instruction, condition variable, OpenMP `single` directive, etc., the sync is identified either by the IR instruction or the name of the function invoked. However, there are currently several cases where Contech does not detect a parallel program's synchronization such as, if the program implements its synchronization through inline assembly, OS signals, or classic synchronization algorithms (e.g., Dekker's algorithm). The synchronization action is recorded with three elements: the address of the action, the order of the action

with respect to other synchronization actions on this address, and time stamps from before/after the action. Where possible, the synchronization action is also tagged with a type, such that it is possible to distinguish, when needed, between atomic instructions, condition variables, etc.

A program that has been instrumented with the front end will output an event list when it runs. Events from the same thread are stored in the event list in program order (not the microarchitectural order from out-of-order processors or memory consistency). Events generated by running threads are placed into thread-local buffers, which have a default size of 1MB. When a buffer is full, the buffer is queued into a global list of buffers [Ansaloni et al. 2012]. Being a global list of buffers, this gives an order to events from different buffers, which can then be used to order specific events between threads. Each type of event (create, join, sync, or barrier) relies on the global list to capture the order information, such as creator versus created or lock acquisition order. When an event with an ordering constraint is generated, the current buffer's contents are forcibly queued to the global list before the thread continues, thus preserving the order.

The runtime framework ensures that synchronizing events are placed in the list according to their partial ordering. For example, the order of two acquire events for the same lock will reflect the execution order of the actual lock acquisitions, yet an acquire by a different thread for a different lock will not necessarily appear earlier in the event list even if it was globally acquired first. Effectively, the event list reflects the partially ordered nature of the parallel program. Furthermore, the interleaving of critical sections observed during an execution of the program is assumed to be fixed. In a single task graph, we are unable to determine whether certain processors could have legally entered critical sections in a different order. To preserve the order, some events have an ordering requirement with other events and preserving this ordering requires globally queuing the event early. In Section 6.1, we will remove this explicit ordering requirement in order to improve the instrumentation performance. When a buffer is queued early, the amount of data in the buffer can be significantly less than the buffer's capacity. Furthermore, whenever a buffer is queued, a new buffer must be allocated to continue to record events in that context. With the runtime system already allocating a new buffer, if the stored data is small, a buffer is allocated for this data, which is copied into the new buffer and allows the full sized buffer to be reused.

Buffers in the global list are written out by a dedicated background thread. This design minimizes the overhead in the running threads to only that required to record an event. Queuing buffers incurs additional overhead, so programs that require more frequent queuing (e.g., have more synchronization events) run more slowly. In Section 6.1, we will discuss how to avoid this overhead. Should the queued event lists reach the threshold of the system's memory limit (default of 90%), the runtime will suspend further execution until all queued buffers have been written to disk. The runtime will also insert events for each suspended thread to indicate that the thread spent time paused.

4.2. Middle Layer

The middle layer processes the event list generated by a front end to convert this trace into a Contech task graph, following Algorithm 1. Contiguous streams of basic blocks in the event list are combined into tasks. Each work task accumulates a list of basic block IDs and memory accesses. When the middle layer encounters a synchronizing event, it closes the current task for the context and determines dependencies between other synchronizing events in the task graph. The middle layer considers the four types of dependency tasks: creates, joins, syncs, and barriers. To avoid any disruption of the running program, the middle layer does not process the trace until the instrumented program has completed. As with the front end, the middle layer also utilizes a

ALGORITHM 1: Assign Events to Tasks

input : Event List
output : Task Graph

```
for e ∈ EventList do
    if e.Type == Basic Block Action then
        if Context[e.context_id].getCurrentTask ().getType () ! = e.Type then
            Context[e.context_id].makeNewTask (e.Type);
        end
        Context[e.context_id].getCurrentTask ().append (e);
    end
    else
        Context[e.context_id].makeNewTask (e.Type);
        // Find other predecessors to the new task
    end
end
```

background thread to write tasks that are complete. In this component, tasks are complete when all predecessors and successors have been identified, and not necessarily when the middle layer has begun processing a successor task. Prior to writing out each complete task, the serialized data of the task is passed through a compression library, and by compressing tasks individually the graph representation supports efficient random accesses. Having prepared the task graph, the middle layer also records a breadth-first traversal of the task graph, starting at Task 0:0 and following the task dependencies. Given multiple possible tasks to visit next, the middle layer checks the time stamps, if present, and selects the task with the oldest start time. This traversal provides a common order for most analysis tools.

4.3. Back End Tools

The previously mentioned components of Contech work together to produce a rich representation of the execution of a parallel program in the form of a Contech task graph. The goal of Contech is to support a wide variety of back end tools that can characterize the behavior of a program and/or collect metrics by analyzing a task graph. The task graph file contains three elements: the debugging information, the tasks of the graph, and the table of contents that provides a breadth-first traversal and random access to the tasks. Back end tools' traversal of the task graph is supported by Contech's task library API, which is written in C++11. The first step is to instantiate a `TaskGraph` from the file.

```
TaskGraph* tg = TaskGraph::initFromFile(fileName);
```

Tools will commonly iterate through each task in the graph, following the breadth-first traversal. A tool can also request a specific task using `getTaskById(TaskId)`.

```
while (Task* currentTask = tg->getNextTask()).
```

The task library API enables programs to access the features of the task graph, and enables iteration via a set of classes over the components of interest from a task graph: tasks, basic blocks, and memory operations. Commonly accessed properties are the type of task, as well as the tasks start and end times.

```
switch(currentTask->getType()).
```

Table I. System Configuration for Contech Measurements

Intel Xeon X5670	
# of processors	2
Cores per processor	6
Hyperthreading	two-way
Clock speed	2.93GHz
Last level cache size	12MB
Main memory size	48GB

Most of the data in a Contech task graph is part of the work tasks. Each work task contains a sequence of basic blocks executed, as well as the memory operations performed by the instrumented code.

```
auto bba = currentTask->getBasicBlockActions();
  for (auto bbIt = bba.begin(), bbEt = bba.end(); bbIt != bbEt; ++bbIt){
      BasicBlockAction bb = *bbIt;
```

The API also exposes "debugging information" to back ends, such as function names, file names, and line numbers. This information is contained in `TaskGraphInfo` class.

```
TaskGraphInfo* tgi = tg->getTaskGraphInfo();
```

Most of the debug information is currently associated with basic blocks, in the `BasicBlockInfo` class.

```
auto bbi = tgi->getBasicBlockInfo((uint)bb.basic_block_id);
```

In analyzing a task graph, a back end might simulate branch behavior, caches and coherence, analyzing synchronization usage, or finding data races.

5. PROGRAM ANALYSIS

In this section, we present three example back ends to demonstrate the benefits of the Contech task graph representation in analyzing the behavior of parallel programs from the PARSEC [Bienia 2011] and NAS [Jin et al. 1999] benchmark suites. The instrumented programs run on a cluster of machines configured based on Table I. Each benchmark was set to run with 16 threads; however, most benchmarks spend significant time at lower degrees of parallelism due to serial sections, locks, or other algorithmic-based decisions. While Contech has support for recording a Region of Interest (ROI), all results in this work are based on tracing the whole program's execution.

Each analysis is applied to a task graph, representing a single instance of the program's execution. Similar to other dynamic analyses, improving the quality of results in an analysis with Contech may require collecting and analyzing multiple task graphs to provide different traces of the program's execution. Most analyses are deterministic and so repeated invocations will not generate different results. For example, the same task graph will always give the same sequence of memory accesses, when following the same partial ordering of all tasks.

5.1. Program Parallelism Over Time

Figure 6 shows the level of parallelism (i.e., unblocked contexts) in the PARSEC and NAS benchmarks across the execution time (in CPU cycles). For display purposes, the results have been downsampled from the original measurements and we have selected examples of different common patterns exhibited by the benchmarks, which could complement improvements discussed in prior work [Poovey et al. 2011]. This is a model of the actual parallelism of the program, similar to the measurements enabled

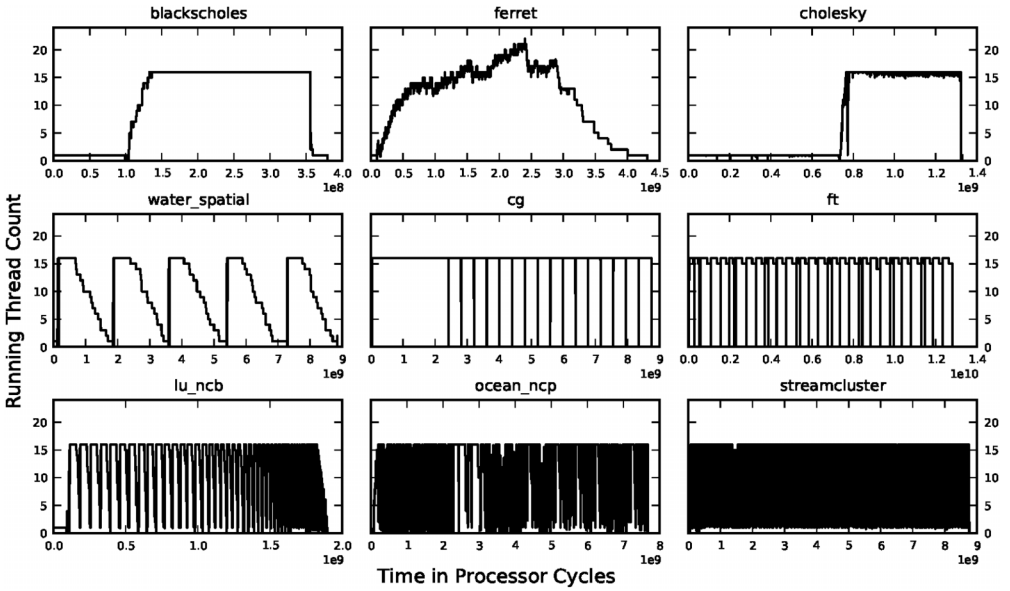

Fig. 6. Parallelism (running threads) in PARSEC benchmarks (simmedium, 16 threads) and NAS (A, 16 threads) over time (processor cycles), results downsampled, and sorted by variability.

by Paraver [Barcelona Supercomputing Center 2015] and ParaMeter [Kulkarni et al. 2009]. Contech uses the start and end times of each task in order to determine the number of active tasks at each unit of time. As a consequence, this is the observed level of parallelism given the system and architecture.

Blackscholes has minimal change in the number of threads executing (once they are spawned), thus exhibiting clear task parallelism. *Ferret* uses thread pools and thus the parallelism changes significantly over time as work passes through the pipeline. *Cholesky* has significant initialization time, which dominates its overall behavior. The remaining benchmarks have increasing quantities of barriers or barrier-like synchronization, which are both indicated by sharp dips or sawtooth patterns and eventually becoming a solid block. The sawtooth pattern of *water_spatial* shows a large difference in runtime among the threads leading to each barrier. The number of barriers varies from *fft* (not pictured), which has three barriers, to *streamcluster* with 13,000 barriers. The NAS workloads, such as *cg* and *ft*, use OpenMP parallel sections, which have an implicit barrier at the end of each section.

5.2. Code Parallelism

Rather than viewing parallelism across time, we can also analyze parallelism across the code in a program. This analysis has been explored in Harmony [Kambadur et al. 2012]. Figure 7 shows how often the different basic blocks were executed at different levels of parallelism (active threads); the active thread count is the count of threads not blocked within the program, even if the operating system has blocked the thread. The basic blocks have three styles of parallelism: serial, mixed, and highly parallel. Most serial code (dark execution counts on the left side) is associated with the initialization and cleanup phases of the program's execution. Execution counts that span the active thread counts indicate that the code is executed with varying parallelism. This variance is usually from two scenarios: first, as the program's threads reach a barrier, the active count decreases while some threads continue executing code. *Water_spatial* is a good

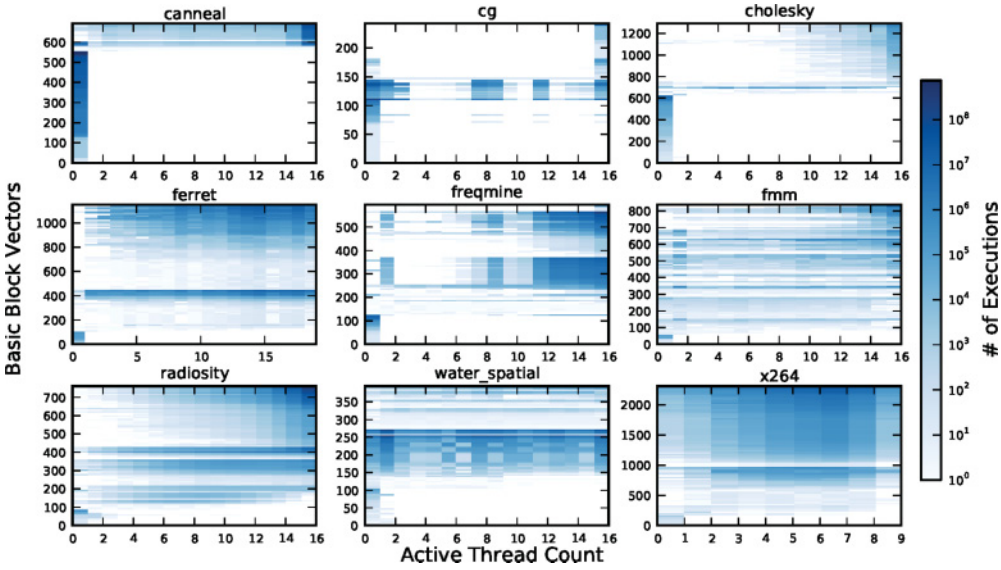

Fig. 7. Heat map (log scale) of active threads across basic block vectors (vectors sorted by level of parallelism).

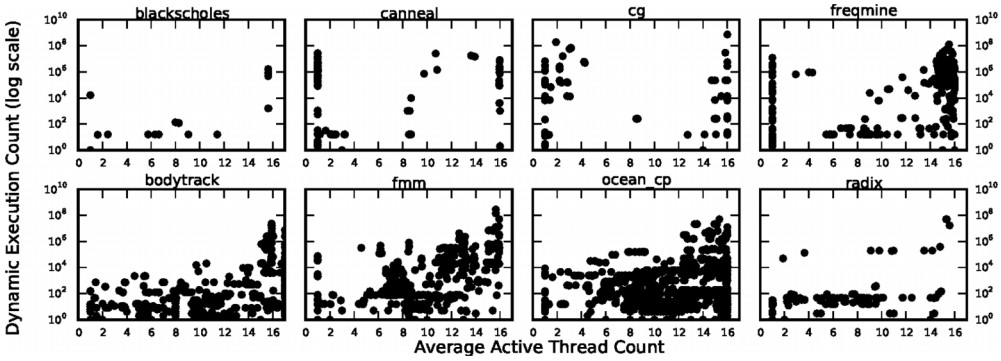

Fig. 8. Average thread count for each basic block plotted against dynamic execution count.

example of this variance. The second scenario occurs when the program uses locks or other synchronization, such that one or more threads are blocked while most are executing. *cholesky* shows this variation in execution with most blocks being executed at the highest parallelism. The final style of parallelism is exhibited by basic blocks that execute when the other threads are never blocked.

In Figure 8, we coalesced the basic block vectors into a single point, averaged across the different thread counts. The basic blocks can be distinguished by average parallelism into the three styles. Across the top row, *blackscholes*, *canneal*, and *cg* have all three distinct partitions of basic-block parallelism styles. Other benchmarks, such as *freqmine*, have a subset of basic blocks that are predominantly used serially without the mixed parallelism blocks being separated from fully parallel blocks. Some workloads do not clearly partition, such as *bodytrack*, *fmm*, and *ocean_cp*. In categorizing the blocks into different styles, this extends the previous analysis that showed different regions of parallelism, such that the parallel regions and serial regions are executing different code, which are therefore amenable to different optimizations for both

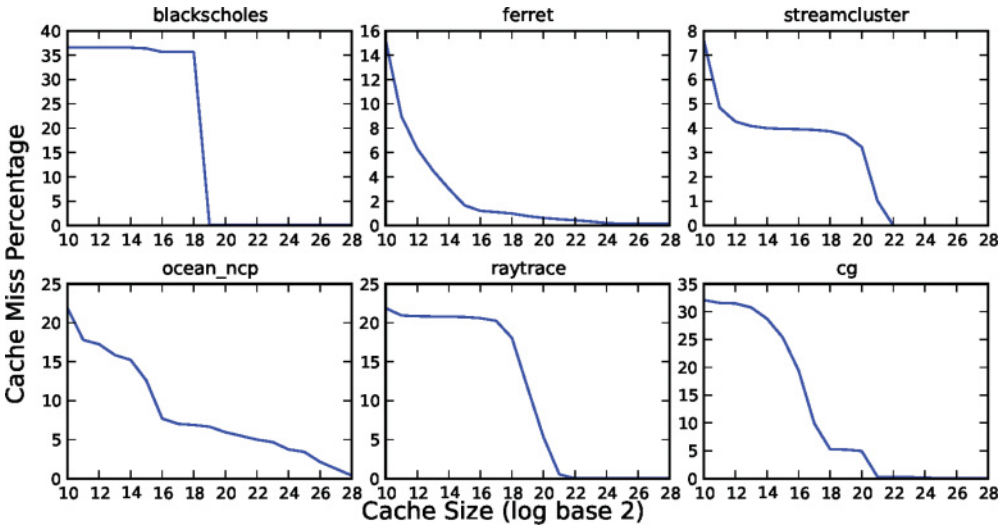

Fig. 9. Miss rate for selected PARSEC, SPLASH-2, and NAS benchmarks with 16 threads, across shared caches sized 1KB to 256MB (2^{10} to 2^{28}).

software and hardware. Finally, the most commonly executed basic blocks can come from any of the three styles and therefore optimizations toward each can have an impact on the program's execution.

5.3. Architectural Models

Contech also provides sufficient information to model architectural components such as caches and branch predictors. In Figure 9, we replay the memory access streams from previously collected task graphs multiple times as we iterate through the range of cache, from 1KB to 256MB. All caches are four-way associative and use least recently used replacement policy (LRU) replacement. This analysis is deterministic and by using the same task graph and breadth-first traversal as stored in the task graph, per Section 4.2, the analysis will operate on identical memory access streams, thus supporting the comparison of the different cache sizes. As with all dynamic analyses, the cache model could require analyzing multiple task graphs from the target program, possibly across diverse inputs, in order to have the necessary confidence in its conclusions.

The cache analysis models show the different characteristics of the benchmark's memory accesses and how increasing the cache size impacts the miss rate. Benchmarks such as *blackscholes* and *raytrace* show minimal improvement from increased cache size until a component of the working set entirely fits in the cache, with significant reduction in miss rates, which contrasts with *ferret* and *ocean_ncp* where the miss rate is continually decreasing from larger caches. *Streamcluster* and *cg* are workloads with multiple plateaus from different components of the working set dominating the cache miss rate.

Choosing a single cache size for each benchmark, we can see further detail about the cache misses. Table II shows the percentage of total cache misses from a single instruction and the function containing that instruction. The table also contains the results tracking each cache miss back to the original allocation of that memory. These results demonstrate two things. First, a significant number of benchmarks have most cache misses from a single memory instruction (either load or store), ranging as high as 81% from a static instruction. Second, most benchmarks allocate large blocks of

Table II. Most Common Missing Instructions and Allocations in PARSEC and NAS Benchmarks

Benchmark	Cache Size Tested (KB)	Function Containing Memory Op	Percentage of Total Misses	Allocation Site (BBID)	Approx. Allocation Size	Percentage of Total Misses
barnes	16	walksub	52.85	113	3.44MB	8.92
blackscholes	256	bs_thread	19.77	67	320KB	99.83
bodytrack	32	ImageMeasurements::...	40.53	57	300KB	12.23
canneal	8	std::_Rb_tree¡std::	23.92	953	18.46MB	11.52
cg	8	conj_grad_._omp_fn.6	78.08		—Fortran—	
cholesky	8	OneMatmat	3.81	153	7.05MB	18.16
dedup	4	rabinseg	20.22	1022	128MB	39.91
ep	512	vranlc_	49.90		—Fortran—	
ferret	32	LSH_query_bootstrap	28.01	604	11.37MB	29.10
fft	8	FFT1DOnce	28.21	61	64.07MB	47.67
fluidanimate	32	ComputeForcesMT	16.29	156	360KB	3.57
fmm	8	VListInteraction	47.19	1409	6.5MB	19.51
freqmine	4	.omp_microtask.3	5.88	1223	10MB	24.99
ft	64	fftz2_	18.90		—Fortran—	
is	32	.omp_microtask.5	62.45		Stack Allocations	
lu_cb	4	lu	40.79	62	513KB	6.43
lu_ncb	64	lu	69.87	39	8MB	99.99
mg	16	resid_._omp_fn.9	9.04		—Fortran—	
ocean_cp	8	jacobcalc2	3.08	575	10.98MB	20.38
ocean_ncp	16	laplacalc	16.65	109	1.95GB	20.46
radiosity	16	traverse_subtree	19.91	982	11.81MB	55.09
radix	64	slave_sort	61.98	54	128MB	46.79
raytrace	128	SphPeIntersect	21.75	1185	32MB	99.22
streamcluster	4	pgain	81.31	340	2MB	84.06
swaptions	16	HJM_SimPath_For	25.69	126	15.13KB	2.06
volrend	16	Trace_Ray	12.04	690	1.64MB	27.87
water_spatial	4	INTERF	13.68	595	80B	2.83
×264	16	block_residual_write	3.26	4789	64KB	4.22

memory. For some programs, such as *blackscholes*, *lu_ncb*, and *raytrace*, almost all misses are from instructions accessing the one single allocation. With *blackscholes*, this allocation can be completely cached by increasing the size by one step and the miss rate in Figure 9 reflects this. In contrast, *ocean_ncp* uses several 2GB allocations, such that 20% or fewer of the total misses are made to each allocated space. These results show how Contech can be used to assist programmers to pinpoint the instruction(s) causing cache misses, as well as link the misses back to the memory allocations and hence the data structures that are not being cached.

6. EXTENSIONS TO THE CONTECH FRONT END

Our goal in building our own front end rather than using existing front ends was to create one with significantly reduced runtime overhead. Low overhead is particularly important when attempting to characterize parallel workloads where the relative overlap and ordering of parallel tasks can be skewed significantly by high overheads [Song and Lee 2014]. This section covers several of the features that have been included in the front end's runtime, their performance, and a comparison with Pin's performance.

6.1. Optimizations to Contech's Implementation

Reducing the Size of the Event Queue: The baseline implementation recorded each basic block event as a 4-tuple of 4-byte fields: context ID, type of event, basic block ID, and number of memory accesses in the basic block. Given how fundamental basic block events are to execution, these values constitute a significant fraction of the event list and even small improvements can have significant impact on the event queue size.

All events in a queued buffer originated from the same context, so each event is carrying redundant information. When the background thread processes a buffer, it adds an additional event to the stream that indicates the next set of bytes all originate from the same context. When the event list is deserialized back into events, the library recognizes these special buffer events and reconstitutes the context ID for each event.

Basic blocks also have a static number of memory accesses known at compile time. Each memory access in a basic block is always of the same type and size. Static information only needs to be stored once for the program and not for each dynamic instance. The compiler pass records this information about each basic block and at link time, this information is added to the binary so that it can be prepended to the event list, such that each memory access no longer needs to explicitly store its type and size information.

When the parallel program is instrumented for 32-bit architectures and 64-bit Intel architectures, memory addresses are stored as 48bits, which is at or above the current limit for virtual addresses [Intel 2014; Larus 1990]. For practical purposes, 32bits each is excessive for the type of event and basic block ID. These two values are combined together into a single 32-bit value. All together, these improvements reduce the space for each basic block event by 75%, when compared to the naïve approach. A further extension would be applying compression to the stream of basic block IDs and memory operations. However, this would impose additional CPU overhead that may not be mitigated by the reductions in memory usage.

Ordering Constraints of Synchronization: Even with the reduction in buffer size, the performance overhead in synchronization-heavy workloads is still significant. Each synchronization event (e.g., program's mutex) forces the current buffer to be queued, and the queue is protected by an internal lock in the Contech runtime. Thus, independent locks in a parallel program are made dependent and their accesses serialize. The middle layer relies on the order of the sync events in the event list to order the sync tasks in the graph.

The global lock provides an ordering of events that could also be reconstituted in the middle layer. Each synchronization event is annotated with a global ordering number (a.k.a. a *ticket*), and the synchronization events are changed to no longer require a buffer flush for ordering purposes. Thus, when a synchronization event occurs, it requests the next *ticket* and execution continues. The middle layer sorts the sync events corresponding to each address using the *tickets* to provide an ordering, so that it can create the appropriate dependencies between sync tasks in the task graph. While this change targeted sync events, the impact of changing one ordering requirement to be implicit effectively forced all of the original event orderings to either use *tickets* or to no longer require a particular order to events. This ordering constraint technically provides a global order to all synchronization events (e.g., lock acquires of different locks have an order); however, this order does not represent program behavior. A further extension to this improvement would be to use separate ticket lists for each lock in the program. Rerun [Hower and Hill 2008] took a similar approach by adding sequential "time stamps" when ordering was required between otherwise independent instruction streams.

Buffer Overflow: When writing events to the buffer, the system needs to ensure that the buffer does not overflow. As the runtime system cannot span a basic block (or other event) across two buffers, the system must check the quantity of data added thus far before inserting a new event. Each check verifies whether a certain margin of space is still available in the buffer (default of 1KB). As each check establishes that sufficient space is still available in the buffer for multiple basic blocks, many of the checks would be redundant and are not inserted into the code. Basic blocks with a large number of memory operations always have a buffer overflow check to ensure that the event for this basic block will not exceed the margin of buffer safety. Having reduced the number of buffer overflow checks in the instrumented code, the remaining checks for buffer overflow account for around 10% of the total instrumented execution time. While other instrumentation approaches have proposed canary values [Zhao et al. 2008] or guard pages [Upton et al. 2009], the additional gains would be minimal versus the added complexity.

Position Value: To simplify the instrumentation design, recording and serializing a basic block event consists of the basic block header followed by separate writes for each memory access. Each is a separate function invocation that is inlined. In order to minimize the memory accesses, the functions are written to pass the buffer pointer between the instrumentation calls in each basic block, which is nonobvious for the compiler to identify. This pointer is also passed to the buffer overflow checks, thus requiring the current buffer position to be computed only once per basic block. The result of this optimization can be seen in Figure 5 where registers `%rcx` and `%rax` are used by the instrumentation instructions, storing the thread-local buffer pointer and its position, respectively.

6.2. Distributed Memory Support

While the focus of this work has been on shared memory programs, Contech also supports distributed memory programs that use MPI to communicate. Each instance of a distributed program generates a separate trace. The middle layer can then merge the traces together into a single task graph. The task graph has an empty task 0 added as the root that will create the instances of the parallel program. In order to maintain the address space separation, each memory address is prepended with the instance of the program (i.e., the rank), thus treating the program as if it used a Partitioned Global Address Space (PGAS). As sending and receiving data is an explicit operation, the ordering of sends and receives is represented as *sync* tasks, along with additional memory operations to denote the transfer of data between the two address spaces.

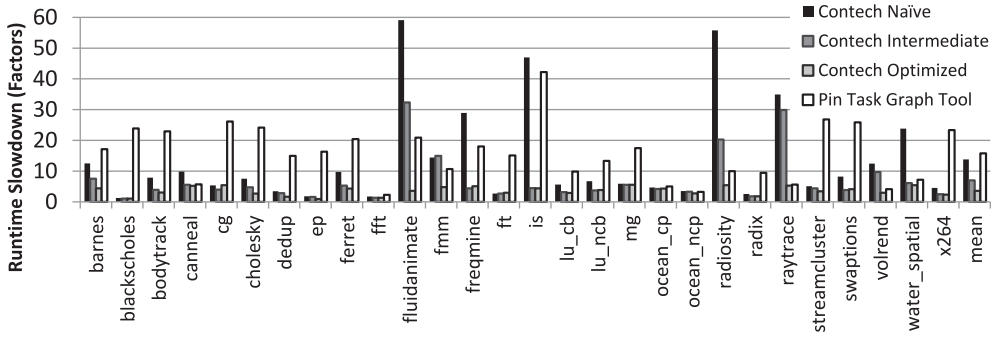

Fig. 10. Comparison of Contech and Pin runtime slowdown for PARSEC and SPLASH (simmedium, 16 threads) and NAS (A, 16 threads).

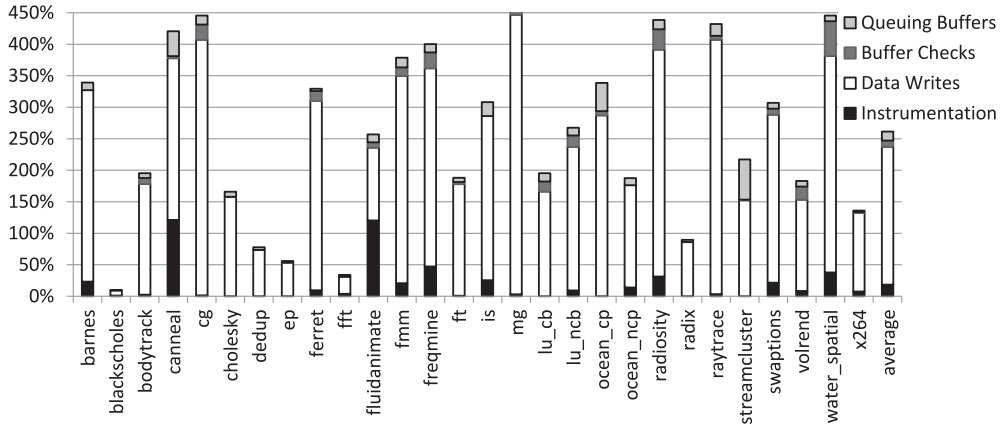

Fig. 11. Overhead increase from runtime instrumentation over uninstrumented programs.

6.3. Performance of Contech Front End

This section presents performance measurements from running the PARSEC and NAS benchmark suites on the cluster of machines configured based on Table I. Figure 10 shows the performance of three Contech implementations using the features described in Section 6.1, along with a comparable Pin tool implementation (see Section 6.4). The naïve implementation is outlined in Section 4.1. The intermediate implementation is improved by reducing the event queue size. The optimized implementation includes the other performance techniques. The original, naïve implementation of the Contech runtime had significant overhead for a majority of the PARSEC benchmarks. Taking advantage of compiler and architectural knowledge to change the structure of events, the runtime overhead was significantly reduced; however, a small number of benchmarks, such as *fluidanimate*, *radiosity* and *raytrace*, still exhibited higher overhead than the remaining benchmarks. These benchmarks make significant use of synchronization. By changing how the synchronization is handled in the runtime, the overhead is reduced and exhibits little variability between benchmarks. The final slowdown is between 1× and 5.5× with an average of 3.6×.

By selectively disabling parts of the runtime, we can measure the overhead imposed by each component of Contech's runtime across all benchmarks. We averaged five runs and compared the result against the baseline execution time. Figure 11 shows the runtime increase with each instrumentation component selectively added in, split

Table III. System Configuration for Contech Measurements on ARM

NVIDIA Tegra K1	
Processor	ARM Cortex A15 r3p3
# of processors	1
Cores per processor	4
Clock speed	2.3GHz
Last level cache size	2MB
Main memory size	2GB

Table IV. Runtime Overhead for PARSEC (simsmall) with Clang and Optimized Contech (without lto) on ARM

Benchmark	Run Time Slowdown
blackscholes	4.07
cholesky	31.47
dedup	6.41
volrend	11.03

into four categories: the instrumentation instructions, the cost of writing the events to memory, the overhead of checking the buffer position, and the time to queue full thread-local buffers. For most workloads, the processors have sufficient resources to execute the instrumentation with low overhead (20%). Writing the events into the thread-local buffer still takes the majority of execution time even after the previous improvements. We changed these writes to use nontemporal, write-combining writes (i.e., SSE2's movnti); however, this results in a minimal reduction to this overhead. With a high rate of memory traffic and regular nature of the instrumentation's memory writes, there is also little benefit to prefetching the thread-local buffer. The remaining checks for overflowing the thread-local buffer only account for 10% of the execution time, and the program's branch prediction rate improves with the instrumentation. The time required to allocate a new 1MB thread-local buffer and queue the existing one into the global queue takes around 14% of the execution time; programs such as *swaptions* have higher queuing overhead due to the frequent memory allocations in the original program. Nearly all of Contech's overhead comes from the impact to the data cache from the thread-local buffers forcing out application data.

The time to compile each application increases by an average of 68%. Two compiler passes are required for each source file, as Clang does not support dynamic loading LLVM passes. Integrating the pass into Clang would avoid this issue. The instrumentation generates data at an average rate of 5GB/s for each benchmark (between 1.6 and 8.1GB/s). This data is being written out in the background while the program is executing; however, most hard disks do not support 5GB/s transfer rates, so the event list buffers in memory during program execution. As mentioned earlier, should the queued buffers reach the memory usage threshold, the instrumentation suspends the program's execution until there are no queued buffers. Moving this background writing to multiple threads could enable online analysis and compression [Ha et al. 2009], yet greatly increase the processing requirements of the running machine.

To demonstrate support for other ISAs, we executed a subset of PARSEC benchmarks with Contech instrumentation using an ARM-based system (see Table III). Given the small size of the system, these workloads used the *simsmall* input, with the slowdowns in Table IV. These slowdowns are higher than those observed on $\times$86 due to the following reasons: limited inlining of instrumentation and limited microarchitectural resources. An LLVM bug prevents the link-time optimizer from working properly on ARM; the most common instrumentation calls are marked force_inline, but the instrumentation is not being optimized to the same degree as on $\times$86 and therefore increases

the slowdown. In addition, the ARM processor does not have the same quantity of spare microarchitectural resources for the instrumentation, and only generates data at less than one-quarter the Intel processor's rate. The only change to the Contech framework required for ARM was adding one line of assembly to access the time stamp counter (PMCCNTR).

6.4. Comparison with Pin

Pin is a widely used, high performing instrumentation tool that has support for collecting similar data to Contech's task graphs. We searched prior work and found several tools that collect subsets of the data recorded by Contech: recording a memory trace alone (ignoring basic blocks, as well as attributes of the memory accesses) ranged between 2× and 9× slowdown [Bach et al. 2010; Laurenzano et al. 2012], recording the memory trace with attributes and basic block information averaged a 16× slowdown [Zhao et al. 2008], and recording the program's DAG and instruction count incurs 2×–10× slowdown [He et al. 2010]. As Contech's instrumentation is more extensive than these tools, we implemented our own Pin tool that is functionally equivalent to Contech's instrumentation, for Pin 2.13 that uses Pin's Fast Buffering APIs [Upton et al. 2009] to record basic block IDs and memory operations, with read or write and size attributes, along with events for synchronization actions. We incorporated optimizations to the Pin tool including those in Section 6.1 such that the performance would be comparable to prior work.

Running the PARSEC and NAS benchmarks using Pin, we recorded the execution time to estimate the overhead imposed by Pin. The results are the average of five runs, and the measurements are for the whole program's execution. Figure 10 shows the runtime slowdown for the benchmarks, comparing Contech's front end versions with Pin. This set of measurements show that the current implementation of Contech's front end performs better than a Pin-based implementation does on all workloads. The Pin slowdowns also show significant variation across the PARSEC benchmarks, varying from 2.3× to 42× and averaging 15.8×.

We used the tool *perf* to measure the hardware performance counters to compare Contech's and Pin's instrumentation. Being a dynamic instrumentation tool, Pin significantly impacts the instruction cache performance with an L1I cache miss rate of between 2.2% and 8.0% averaging 4.4% versus Contech's range of 0.1%–7.1% with an average of 0.7%. The uninstrumented code's instruction cache miss rates ranged from 0.02% to 4.0% with an average of 0.5%.

Figure 12 shows the increase in architectural events from Contech's and Pin's instrumentation versus the baseline programs. In all cases, the instrumentation has increased the operations and instructions executed by the programs. Along with the measurements in Figure 11 where the instrumentation instructions were shown to have low overhead, the runtime cost of the additional operations recorded with *perf* is primarily the increased cache pressure.

7. CONCLUSION

Contech provides architects, programmers, and automated tools with a novel task graph representation that enables a rich, meaningful analysis of parallel programs. This representation can be collected from programs with arbitrary parallelism, based on the architecture independent design, allowing the analysis of C/C++/Fortran programs that use pthreads, OpenMP, or MPI on ×86 or ARM ISAs. Three example parallel program analyses demonstrate the diversity of information contained within Contech task graphs and its possible usage in understanding programs. Finally, we have shown the details of how the framework is implemented in order to achieve state-of-the-art

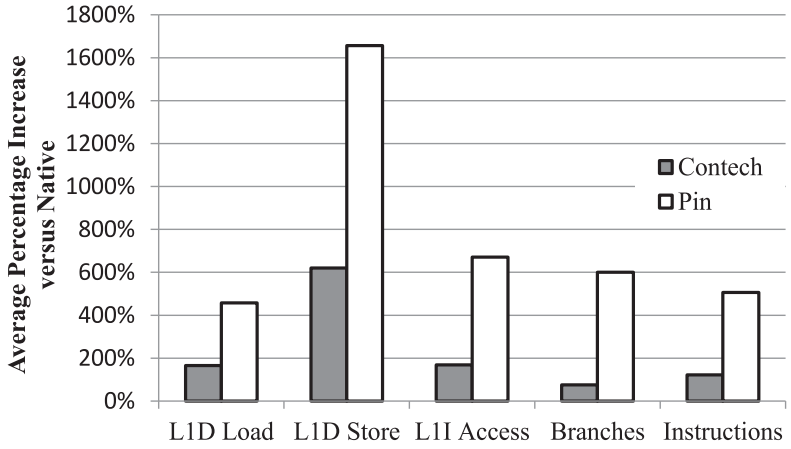

Fig. 12. Average increase of architectural events over uninstrumented execution.

performance, resulting in an average of 3.6× slowdown of instrumented parallel programs.

In future work, we plan to extend Contech to support Cilk using the Clang-based project that is currently under development. We are also investigating how to viably instrument blocks of inline assembly that contain function calls of interest, as well as identifying synchronization from atomic instructions. We are developing additional parallel program analyses that use Contech task graphs. Finally, we are always looking at other opportunities to further improve the instrumentation's performance.

REFERENCES

V. S. Adve and R. Sakellariou. 2001. Compiler synthesis of task graphs for parallel program performance prediction. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers (LCPC'00)*. Springer-Verlag, London, UK, 208–226. http://dl.acm.org/citation.cfm?id=645678.663959.

V. S. Adve and M. K. Vernon. 2004. Parallel program performance prediction using deterministic task graph analysis. *ACM Trans. Comput. Syst.* 22, 1 (Feb. 2004), 94–136. DOI:http://dx.doi.org/10.1145/966785.966788

K. Agrawal, C. E. Leiserson, and J. Sukha. 2010. Executing task graphs using work-stealing. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–12. DOI:http://dx.doi.org/10.1109/IPDPS.2010.5470403

V. A. F. Almeida, I. M. M. Vasconcelos, J. N. C. Árabe, and D. A. Menascé. 1992. Using random task graphs to investigate the potential benefits of heterogeneity in parallel systems. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing (Supercomputing'92)*. IEEE Computer Society, Los Alamitos, CA, 683–691. http://dl.acm.org/citation.cfm?id=147877.148113

D. Ansaloni, W. Binder, A. Heydarnoori, and L. Y. Chen. 2012. Deferred methods: Accelerating dynamic program analysis on multicores. In *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO'12)*. ACM, New York, NY, 242–251. DOI:http://dx.doi.org/10.1145/2259016.2259048

M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, Chi-Keung Luk, G. Lyons, H. Patil, and A. Tal. 2010. Analyzing parallel programs with pin. *Computer* 43, 3 (2010), 34–41. DOI:http://dx.doi.org/10.1109/MC.2010.60

Barcelona Supercomputing Center 2015. *Paraver*. Barcelona Supercomputing Center. http://www.bsc.es/computer-sciences/performance-tools/paraver.

C. J. Beckmann and C. D. Polychronopoulos. 1992. Microarchitecture support for dynamic scheduling of acyclic task graphs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO 25)*. IEEE Computer Society, Los Alamitos, CA, 140–148. http://dl.acm.org/citation.cfm?id=144953.145791.

C. Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

G. E. Blelloch, P. B. Gibbons, Y. Matias, and G. J. Narlikar. 1997. Space-efficient scheduling of parallelism with synchronization variables. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'97)*. ACM, New York, NY, 12–23. DOI: http://dx.doi.org/10.1145/258492.258494

R. D. Blumofe and C. E. Leiserson. 1993. Space-efficient scheduling of multithreaded computations. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC'93)*. ACM, New York, NY, 362–371. DOI: http://dx.doi.org/10.1145/167088.167196

D. Bruening, T. Garnett, and S. Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO'03)*. IEEE Computer Society, Washington, DC, 265–275. http://dl.acm.org/citation.cfm?id=776261.776290.

H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. 2011. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, New York, NY, 337–351. DOI: http://dx.doi.org/10.1145/2043556.2043588

Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero. 2010. Task superscalar: An out-of-order task pipeline. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'43)*. IEEE Computer Society, Washington, DC, 89–100. DOI: http://dx.doi.org/10.1109/MICRO.2010.13

A. Gerasoulis, S. Venugopal, and T. Yang. 1990. Clustering task graphs for message passing architectures. In *Proceedings of the 4th International Conference on Supercomputing (ICS'90)*. ACM, New York, NY, 447–456. DOI: http://dx.doi.org/10.1145/77726.255188

A. Goel, A. Roychoudhury, and T. Mitra. 2003. Compactly representing parallel program executions. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*. ACM, New York, NY, 191–202. DOI: http://dx.doi.org/10.1145/781498.781530

G. Gupta and G. S. Sohi. 2011. Dataflow execution of sequential imperative programs on multicore architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, 59–70. DOI: http://dx.doi.org/10.1145/2155620.2155628

J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. 2009. A concurrent dynamic analysis framework for multicore hardware. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'09)*. ACM, New York, NY, 155–174. DOI: http://dx.doi.org/10.1145/1640089.1640101

Y. He, C. E. Leiserson, and W. M. Leiserson. 2010. The cilkview scalability analyzer. In *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*. ACM, New York, NY, 145–156. DOI: http://dx.doi.org/10.1145/1810479.1810509

D. R. Hower and M. D. Hill. 2008. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*. IEEE Computer Society, Washington, DC, 265–276. DOI: http://dx.doi.org/10.1109/ISCA.2008.26

Intel. 2014. *Intel 64 and IA-32 Architectures Software Developer Manuals*. Intel Corporation, Santa Clara, CA. Retrieved from http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

H. Jin, M. Frumkin, and J. Yan. 1999. *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance*. Technical Report NAS-99-011. NAS.

M. Kambadur, K. Tang, and M. A. Kim. 2012. Harmony: Collection and analysis of parallel block vectors. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*. IEEE Computer Society, Washington, DC, 452–463. http://dl.acm.org/citation.cfm?id=2337159.2337211

M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. 2009. How much parallelism is there in irregular applications?. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)*. ACM, New York, NY, 3–14. DOI: http://dx.doi.org/10.1145/1504176.1504181

S. Kumar, C. J. Hughes, and A. Nguyen. 2007. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. ACM, New York, NY, 162–173. DOI: http://dx.doi.org/10.1145/1250662.1250683

J. R. Larus. 1990. Abstract execution: A technique for efficiently tracing programs. *Softw. Pract. Exper.* 20, 12 (Nov. 1990), 1241–1258. DOI: http://dx.doi.org/10.1002/spe.4380201205

J. R. Larus. 1999. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI'99)*. ACM, New York, NY, 259–269. DOI: http://dx.doi.org/10.1145/301618.301678

C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization:*

Feedback-directed and Runtime Optimization (CGO'04). IEEE Computer Society, Washington, DC, p. 75. http://dl.acm.org/citation.cfm?id=977395.977673.

M. A. Laurenzano, J. Peraza, L. Carrington, A. Tiwari, W. A. Ward, and R. Campbell. 2012. A static binary instrumentation threading model for fast memory trace collection. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis (SCC'12)*. IEEE Computer Society, Washington, DC, 741–745. DOI:http://dx.doi.org/10.1109/SC.Companion.2012.101

D. L. Long and L. A. Clarke. 1989. Task interaction graphs for concurrency analysis. In *Proceedings of the 11th International Conference on Software Engineering (ICSE'89)*. ACM, New York, NY, 44–52. DOI:http://dx.doi.org/10.1145/74587.74592

C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, NY, 190–200. DOI:http://dx.doi.org/10.1145/1065010.1065034

Y.-H. Lyu, D.-Y. Hong, T.-Y. Wu, J.-J. Wu, W.-C. Hsu, P. Liu, and P.-C. Yew. 2014. DBILL: An efficient and retargetable dynamic binary instrumentation framework using llvm backend. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'14)*. ACM, New York, NY, 141–152. DOI:http://dx.doi.org/10.1145/2576195.2576213

M. McCool, J. Reinders, and A. Robison. 2012. *Structured Parallel Programming: Patterns for Efficient Computation* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA.

N. Nethercote and J. Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, New York, NY, 89–100. DOI:http://dx.doi.org/10.1145/1250734.1250746

J. A. Poovey, B. P. Railing, and T. M. Conte. 2011. Parallel pattern detection for architectural improvements. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism (HotPar'11)*. USENIX Association, Berkeley, CA, 12–12. http://dl.acm.org/citation.cfm?id=2001252.2001264

B. P. Railing and E. R. Hein. 2015. *Contech*. Georgia Institute of Technology. https://github.com/bprail/contech

A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero. 2011. Trace-driven simulation of multithreaded applications. In *2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 87–96. DOI:http://dx.doi.org/10.1109/ISPASS.2011.5762718

V. Sarkar and B. Simons. 1994. Parallel program graphs and their classification. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, London, UK, 633–655. http://dl.acm.org/citation.cfm?id=645671.665396.

Y. W. Song and Y.-H. Lee. 2014. On the existence of probe effect in multi-threaded embedded programs. In *Proceedings of the 14th International Conference on Embedded Software (EMSOFT'14)*. ACM, New York, NY, Article 18, 9 pages. DOI:http://dx.doi.org/10.1145/2656045.2656062

S. Sridharan, G. Gupta, and G. S. Sohi. 2014. Adaptive, efficient, parallel execution of parallel programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, New York, NY, 169–180. DOI:http://dx.doi.org/10.1145/2594291.2594292

S. Tallam and R. Gupta. 2007. Unified control flow and data dependence traces. *ACM Trans. Archit. Code Optim.* 4, 3, Article 19 (Sept. 2007). DOI:http://dx.doi.org/10.1145/1275937.1275943

D. Upton, K. Hazelwood, R. Cohn, and G. Lueck. 2009. Improving instrumentation speed via buffering. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA'09)*. ACM, New York, NY, 52–61. DOI:http://dx.doi.org/10.1145/1791194.1791202

H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. 2013. Analysis of dependence tracking algorithms for task dataflow execution. *ACM Trans. Archit. Code Optim.* 10, 4, Article 61 (Dec. 2013), 24 pages. DOI:http://dx.doi.org/10.1145/2555289.2555316

R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and Christos. Kozyrakis. 2013. Locality-aware task management for unstructured parallelism: A quantitative limit study. In *Proceedings of the 25th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'13)*. ACM, New York, NY, 315–325. DOI:http://dx.doi.org/10.1145/2486159.2486175

Q. Zhao, I. Cutcutache, and W.-F. Wong. 2008. Pipa: Pipelined profiling and analysis on multi-core systems. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'08)*. ACM, New York, NY, 185–194. DOI:http://dx.doi.org/10.1145/1356058.1356083

Q. Zhao, I. Cutcutache, and W.-F. Wong. 2010. PiPA: Pipelined profiling and analysis on multicore systems. *ACM Trans. Archit. Code Optim.* 7, 3, Article 13 (Dec. 2010), 29 pages. DOI:http://dx.doi.org/10.1145/1880037.1880038

Received January 2015; revised May 2015; accepted May 2015