

# Automatic Run-Time Extraction of Communication Graphs From Multithreaded Applications

Ai-Hsin Liu and Robert P. Dick

Electrical Engineering and Computer Science Department  
Northwestern University  
2145 Sheridan Road  
Evanston, Illinois 60208

ai-hsin-liu@northwestern.edu dickrp@eecs.northwestern.edu

## ABSTRACT

Embedded system synthesis, multiprocessor synthesis, and thread assignment policy design all require detailed knowledge of the run-time communication patterns among different threads or processes. Researchers have commonly relied on manual estimation, compile-time analysis, or synthetic benchmarks when developing and evaluating synthesis algorithms and thread assignment policies. In a more ideal world, it would be possible to quickly and easily determine the run-time communication properties of large commercial and academic multithreaded applications. This article describes a fully-automated method of extracting run-time communication graphs from multithreaded applications.

The resulting graphs may be used to better understand, design, and synthesize application-specific hardware-software systems. The proposed graph extraction method is implemented as a module within the Simics multiprocessor simulator. It presently supports the analysis of arbitrary multithreaded applications running on the Linux operating system. This software is called CETA. It is freely available for academic and non-profit use.

**Categories and Subject Descriptors:** C.1.4 [Processor Architectures]: Parallel Architectures; C.5.4 [Computer System Implementation]: VLSI Systems

**General Terms:** Design, Algorithms, Performance, Measurement

**Keywords:** Multithread, Communication, Task graph, Benchmarks, Synthesis, Run-Time

## 1. INTRODUCTION

Many embedded system and multiprocessor synthesis algorithms start from specifications in the forms of graphs [7,8,11,13,18,19]. Developing and validating such algorithms depends on access to graph representations of the communication among different tasks, processes, or threads. However, as explained in the following paragraphs, large commercial graph-based application specifications are scarce in the system synthesis community. This article describes an automated method of extracting run-time communication graphs from real multithreaded applications.

Easy-to-use, automated extraction of run-time communication dataflow graphs sits between two research communities. Com-

---

This work was supported in part by the NSF under award CNS-0347941.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*CODES+ISSS'06*, October 22–25, 2006, Seoul, Korea.

Copyright 2006 ACM 1-59593-370-0/06/0010 ...\$5.00.

puter architects are generally familiar with the simulation techniques necessary for extracting these graphs. However, few architects need these data in their research. Application-specific system synthesis researchers most need these graphs. However, developing automated methods to extract them would require learning tools that are rarely used in their primary research area. The proposed technique builds on work in the computer architecture community but seeks to solve a pressing problem for those in the embedded system and multiprocessor synthesis communities.

In an ideal world, embedded system and multiprocessor synthesis algorithms would be developed and evaluated using large industrial application specifications. However, in practice, there are a number of legal, language, and economic barriers to this goal. Companies are generally reluctant to release even anonymized versions of their embedded system specifications. Doing so can undermine trade secret protection. In addition, this would require effort without immediate benefit to the company. Finally, even if other barriers did not exist, the formats of specifications may vary from company to company, making use for developing and evaluating synthesis algorithms difficult. We are aware of a few cases in which companies released anonymized embedded specifications to synthesis researchers. Those cases required great effort and ultimately did little to publicly validate synthesis techniques because the specifications could not be further distributed due to non-disclosure agreements.

A number of researchers have written or gathered embedded system specifications for use in synthesis. However, producing detailed embedded system specifications is time-consuming. It necessarily reduces the time that can be spent developing and evaluating new synthesis ideas. As a result, there are few hand-built academic synthesis benchmarks, and most of those that exist are small.

Researchers have developed methods of converting from medium-level programming languages such as C and C++ to dataflow graphs [15]. However, these techniques can only extract information available at compile time. In order to capture the run-time dataflow properties of applications, dynamic techniques, of the sort described in this article, are necessary.

Researchers have developed pseudo-random graph generation algorithms for use in scheduling and allocation research [4]. These pseudo-random techniques are meant to produce large graphs that, hopefully, approximate the structures of real applications. Such techniques serve as last resorts when an insufficient number of large real specifications are available. Their use in scores of papers on system synthesis is evidence that the research community could benefit from an easy-to-use method of generating communication graphs from real applications.

Researchers have developed sophisticated processor and system simulation environments [3,6,10,14] that accommodate the execution of unmodified multithreaded software applications on widely-used operating systems (OSs). However, although such simulators execute OS code with fidelity to real machines, they typically operate with no knowledge of the implications of operations occurring within the simulated OSs. We have extended Simics [10], a multiprocessor simulator, so that it supports the operating system

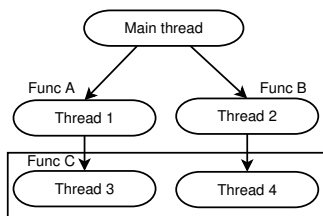


Figure 1: Example multithreaded application.

**Table 1: Process IDs and names**

| PID  | Name           | Description      |
|------|----------------|------------------|
| 0    | swapper        | Ancestor process |
| 1    | init           | Root process     |
| 1052 | thread_example | master           |
| 1053 | thread_example | main             |
| 1054 | thread_example | Thread 1         |
| 1055 | thread_example | Thread 2         |
| 1056 | thread_example | Thread 3         |
| 1057 | thread_example | Thread 4         |

introspection necessary to trace dataflow in multithreaded shared memory applications.

Existing projects have proposed to use simulation, instrumentation, and special-purpose hardware to profile distributed or parallel applications [2,9,16]. However, existing tools have one or more of the following drawbacks when used in multiprocessor synthesis research: dependence on special hardware; intrusive modification of the applications under analysis; dependence on a particular communication protocol, e.g., a specific message passing library; or neglect of communication. Some dynamic instrumentation techniques for memory profiling are promising but have been tailored for use in redesigning applications instead of synthesis [17].

This article describes an easy-to-use tool for extracting runtime communication graphs from multithreaded applications. This tool is called Communication Extraction from Threaded Applications (CETA). It is composed of a Simics module for extracting dataflow information during the simulation of multithreaded Linux applications and a number of scripts that implement graph transformations. The purpose of CETA is to extract application dataflow information in a hardware-independent manner, enabling other tools to determine and optimize the performance of the application and hardware-software implementations. Three output formats are supported for use in application analysis and optimization, multiprocessor synthesis, and thread assignment. CETA supports analysis of applications for which either the source code or executable code is available, and gives the user control over the portions of applications on which dataflow analysis will be used. CETA is freely available for academic and non-profit use at <http://www.eecs.northwestern.edu/~dickrp/projects.html>.

The rest of the paper is organized as follows. Section 2 describes CETA’s use and operation on an example application, Section 3 describes the design and implementation of CETA, Section 4 describes the results of using CETA on multithreaded multimedia applications, and Section 5 concludes the paper.

## 2. ILLUSTRATIVE EXAMPLE

In this section, we illustrate the operation of CETA by describing the production of a communication graph from a multithreaded application. We have selected a small application to ease explanation. Please note that CETA can analyze multithreaded applications of arbitrary complexity and size, as demonstrated in Section 4.

Figure 1 illustrates the order in which an example multithreaded application spawns its five threads. In the `main()` function the main thread creates Thread 1 to execute Function A and Thread 2 to execute Function B. After executing for a short while, Thread 1 and Thread 2 create Thread 3 and Thread 4 in Function A and

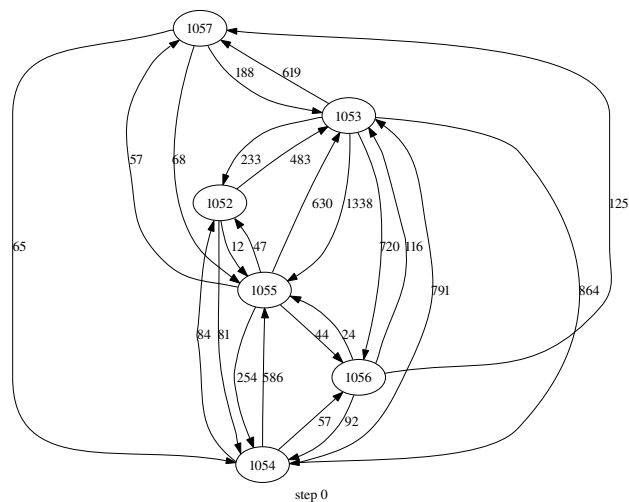


Figure 2: Simplified communication graph for example application.

Function B, respectively. When Thread 3 and Thread 4 are created, they each execute Function C.

In a multithreaded Linux application, each thread is a lightweight processes with its own process identifier (PID). Although threads have individual PIDs, register states, and stack pointers, they typically share memory and file descriptors with other threads. This makes shared memory communication among threads convenient. The `main()` function allocates an array. When a thread creates another, the two threads share the same address space and, therefore, both have access to this array.

### 2.1 Aggregate Communication Graphs

Figure 2 shows the run-time communication graph CETA extracted from the example application shown in Figure 1. In Figure 2, we have omitted OS threads for simplicity. Each node represents a process or thread. The edges indicate the directions in which data are communicated among threads. The edge labels indicate the amount of data communicated between the two connected threads. Each node in this graph is labeled with its corresponding PID. The tasks associated with these threads are listed in Table 1. We will first focus on the threads within the application, then explain the remaining information provided by the CETA communication graph.

The six threads for the example program are shown in Figure 2. The main thread has a PID of 1053 and the master thread, which has a PID of 1052, is used by Pthreads to manage other threads. PIDs 1054 and 1055 correspond to Thread 1 and Thread 2, which are created by the main thread. PIDs 1056 and 1057 correspond to Thread 3 and Thread 4, which are created by Thread 1 and Thread 2, respectively. As shown in Figure 2, PID 1053 communicates a high volume of data to PID 1055 (1338 bytes), PID 1054 (864 bytes), and PID 1056 (720 bytes). The master thread, PID 1052, only communicates with PID 1053, PID 1054, and PID 1055, the main thread and the threads created by it. PID 1056 receives data from the main thread and its parent thread, PID 1054, and sends data to PID 1054. In summary, each thread communicates frequently with the main thread via reads and writes to shared memory.

### 2.2 Phase Partitioned Communication Graphs

Although the communication graph in Figure 2 contains useful information about the communication properties of the application, it does not indicate whether these properties change during execution. For example, some threads may communicate heavily during some phases of execution and other threads may communicate heavily during other phases of execution. This phase-specific information is useful during synthesis or the development of thread

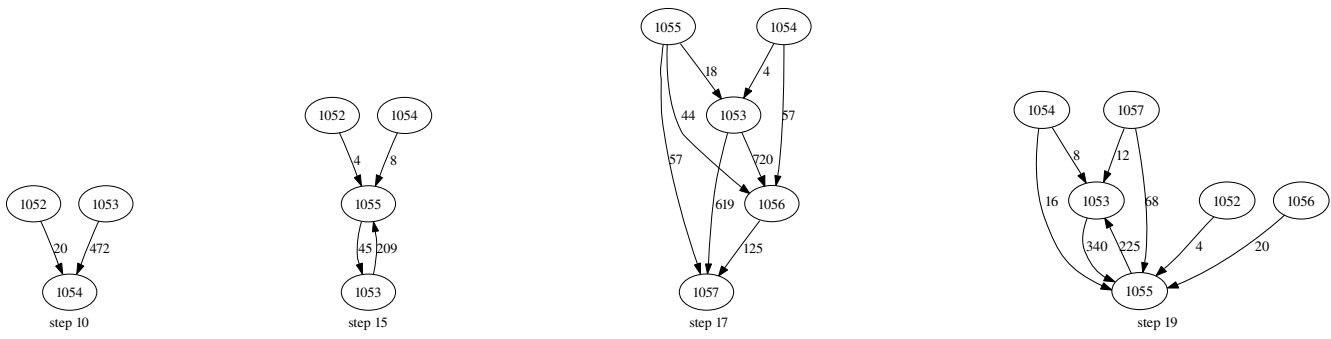


Figure 3: Phases for example multithreaded application.

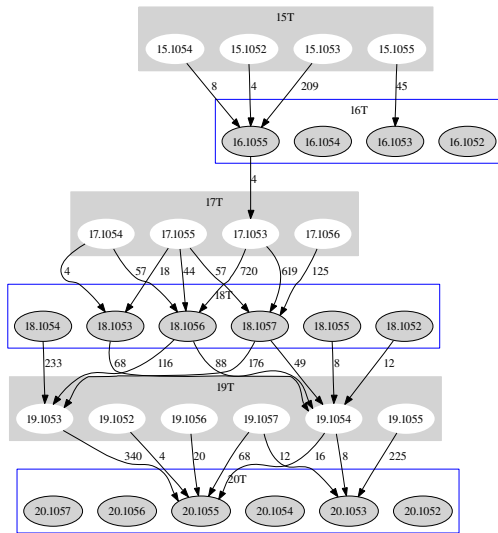


Figure 4: Directed acyclic communication graph for example application.

assignment algorithms. For example, phase-dependent communication profiles can allow the designer or synthesis tool to construct communication networks that minimize the impact of contention. CETA can automatically partition communication graphs with arbitrary, user-specified, granularity.

Figure 3 illustrates the result of CETA partitioning the communication graph for the example application with a period of 5,000 CPU clock cycles, resulting in 21 graphs for the example program. The step duration is provided as a parameter to the CETA module. Here we show four of these graphs, including only Steps 10, 15, 17, and 19. For this example, a label of Step  $n$  denotes the communication graph for cycles  $5,000n$  to  $5,000(n+1) - 1$ . During Step 10 of Figure 3, a significant amount of data flows from PID 1053 to PID 1054 when the `main(.)` function transmits an array of data to Function A. In Step 15, PID 1053 creates PID 1055, within which Function B reads the contents of an array in shared memory. In Step 17, a large amount of data flows from the main thread (PID 1053) to Thread 3 (PID 1056) and Thread 4 (PID 1057). Finally, in Step 19, PIDs 1053 and 1055 communicate often as PID 1055 rejoins with PID 1053. This sort of phase partitioned communication graph can be used to determine the time-varying communication behavior of a multithreaded application, allowing the application to be optimized, or enabling the synthesis of an architecture on which it may run with good performance.

### 2.3 Directed Acyclic Communication Graphs

A number of synthesis algorithms target dataflow applications specified by sets of periodic, acyclic dataflow graphs. This repre-

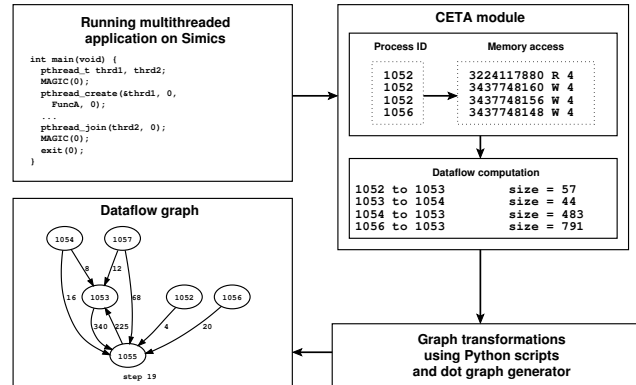


Figure 5: CETA design overview.

sentation is particularly useful for hard and soft real-time applications; by eliminating unbounded loops, it permits bounds on graph execution times. In the case of true hard real-time applications, deriving hardware-software synthesis specifications from run-time communication graphs is unacceptable unless it is possible to guarantee that the worst-case communication volume path was exercised. However, it is appropriate to use such graphs in soft real-time applications, e.g., multimedia consumer electronics.

CETA has the ability to transform the multi-phase communication graphs described in Section 2.2 into directed acyclic communication graphs as shown in Figure 4. This transformation changes the properties of the original graphs only as required by the temporal discretization process. The transformation ensures that all communication occurs between temporally adjacent phases, and travels in the direction of increasing time.

Figure 4 shows the directed acyclic graph derived from steps 15–19 of the example multithreaded application shown in Figure 3. In this figure, Step 15 corresponds to the communication between the phases starting at  $15 \cdot 5,000$  and  $16 \cdot 5,000$  CPU cycles. In Figure 4, tasks 15.1054, 16.1054, 17.1054, and 18.1054 represent the same thread in different time phases. This sort of graph may be used directly within existing real-time system synthesis algorithms, i.e., CETA enables the automated extraction of runtime directed acyclic communication graphs from multithreaded applications of arbitrary complexity.

## 3. DESIGN OF CETA

This section describes the design and implementation of CETA, our system for automatic extraction of run-time communication graphs from threaded applications.

### 3.1 Design Overview

CETA is composed of a Simics 3.0.8 multiprocessor simulator module and a number of Python scripts for data analysis and graph transformation. It extracts run-time communication graphs from

multithreaded applications during simulation. Figure 5 illustrates this extraction process. The first stage in the communication graph extraction process is gathering or compiling an executable for the simulated processor. Simics is then used to simulate the multithreaded application and OS. In the current implementation, CETA simulates a virtual workstation running Red Hat Linux version 7.3, which uses the 2.4.8 Linux kernel. This system has a 20 MHz Pentium 4 processor, and 256 MB of RAM, one 19 GB hard disk, and one IDE CD-ROM. Note that the particular processor used has only second-order effects on application dataflow properties. During simulation CETA traces the flow of data among threads by monitoring reads and writes of shared memory. When the application is finished executing, CETA’s Python scripts are used to transform the data into any of the three graph structures described in Section 2.

CETA’s implementation is necessarily specific to the processor architecture, simulator, and OS in use. However, this is a small concern for two reasons. First, most languages can be compiled to Intel x86 executables, permitting the use of CETA’s current implementation for communication graph extraction. Second, even if simulation on another processor or OS is necessary, many of the techniques presented below will still work. In short, CETA need only be able to determine when context switches, reads, and writes occur.

In the following sections, we describe CETA’s methods of associating PIDs with memory accesses and tracking the flow of data and explain its support transforming communication graphs to different formats.

### 3.2 Process Identifier Tracking

Although instruction processor simulators such as Simics have access to all of the data used by the OSs and applications they simulate, they typically do not understand the meaning of these data. In order to determine the PID associated with each memory access, CETA detects OS context switches and traverses OS data structures to track the current PID.

Whenever the instruction pointer (program counter) is equal to the address of the OSs context switch code (`__switch_to()`) at address `0xc0107410` for Linux 2.4.8 kernel [1], the stack pointer (`esp`) is sure to point to the top of the stack for that function. The `get_task_pid()` function maintains a pointer to a task description structure for the next process on its stack, which includes the new PID. This PID is stored in CETA’s memory for later reference during simulated memory accesses.

### 3.3 Dataflow Analysis

This section describes the method used by CETA to extract communication volumes between pairs of processes.

The CETA module is triggered upon all memory reads and writes. It has access to the PID of the currently-running thread, as described in Section 3.2. In addition, the simulator provides the physical address of the memory access and indicates whether the access is a read or write. Using logical addresses would result in neglected communication because different logical addresses may have the same physical address.

Data flows from Thread A to Thread B if Thread A most recently wrote to a location read by Thread B. If Thread C reads data from the same memory location, data flow from Thread A to Thread C is also tabulated. If Thread B reads the same memory location several times, CETA accumulates the total data volume. Note that this method also allows analysis of data flow in message passing applications. However, if data is copied by a manager thread, it will appear to flow transitively through the manager thread.

In order to permit efficient implementation of communication volume tracking, the Hash Map extension to the Standard Template Library (STL) was used associate each physical address with the PID of its most recent writer. When data at an address are written, the value in the Hash Map is replaced. When data at an address are read, communication from the PID stored in the Hash Map value associated with the address, to the PID of the current process, is tabulated. Communication volumes are accumulated in scalars held in another Hash Map keyed on ordered pairs of PIDs.

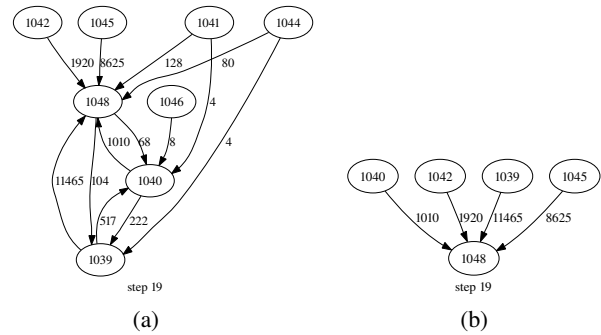


Figure 6: Example graph (a) before and (b) after data volume filtering.

### 3.4 Graph Formats and Transformations

CETA can produce graphs in several formats: (1) aggregate, phase-less, and potentially-cyclic (Section 2.1); (2) phase partitioned and potentially-cyclic (Section 2.2); and (3) phase partitioned and acyclic (Section 2.3). Phase-less graphs can be viewed as a special case of phased graphs in which only one phase exists. Phase partitioning is done within CETA’s Simics module based on a time step interval supplied by the user.

After Simics reports dataflow information, CETA’s Python scripts format these data as directed graphs in the DOT format [5]. This format is straight-forward, general, and suitable for use in synthesis. It may also be converted to human-readable illustrations using the graphviz application.

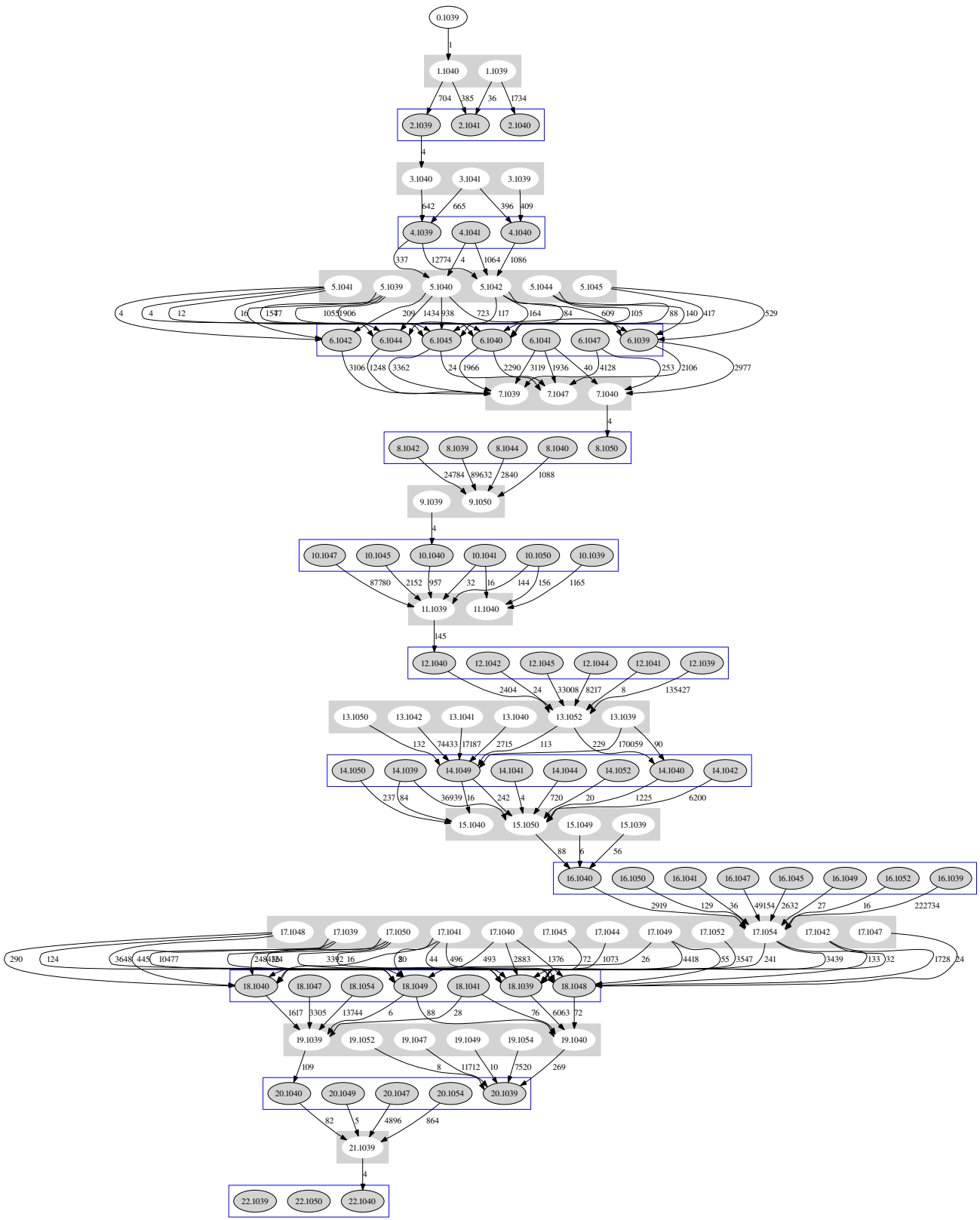
In time partitioning mode, CETA produces a dataflow subgraph for each user-specified time phase. Figures 2 and 3 illustrate the aggregate and phase partitioned versions of communication graphs for the same multithreaded applications. Finally, CETA supports conversion of phase partitioned but potentially cyclic communication graphs into a single acyclic graph (e.g., Figure 4) using the following algorithm. For each phase in the corresponding phase partitioned but potentially cyclic graph, label each vertex ( $v_i$ ) with a symbol indicating the phase ( $p_i$ ). Given that  $\{p_i v_1, p_i v_2, \dots, p_i v_n\}$  are vertices in a phase,  $\{p_{i+1} v_1, p_{i+1} v_2, \dots, p_{i+1} v_n\}$  are the corresponding vertices in the subsequent phase, and  $e[a, b]$  is an edge from vertex  $a$  to vertex  $b$ , replace each edge  $e[p_i v_j, p_i v_k]$  within the same phase with  $e[p_i v_j, p_{i+1} v_k]$  across different phases; i.e., replicate threads in each temporal phase and ensure that communication only flows forward in time.

CETA allows a user to set threshold value for the minimal volume of communicated data that will appear in the communication graph. Starting from Figure 6(a), data volume filtering may be used to eliminate edges with data volumes of less than 1,000 B thereby producing Figure 6(b). CETA also permits filtering by thread PID.

Finally, CETA’s Simics module supports on-line control by simulated applications via Simics magic instructions. Magic instructions are valid instructions for the simulated machine that never occur in normal code. They are detected by the simulator and may be used to trigger events. Magic instruction may be inserted into the high-level language or assembly source code of an application, controlling whether CETA’s Simics module traces or ignores communication. This enables users to focus CETA’s communication graph extraction on particular regions of code.

## 4. CETA EVALUATION

In Section 2, we used a number of graphs to illustrate the operation of CETA. These examples were intentionally kept small to ease explanation. In this section, we present a run-time communication graph automatically extracted from the MPGen benchmark. In addition, statistics are provided for other applications in the ALP-Bench benchmarks suite [12]. For example, the number of vertices are reported in order to illustrate the use of CETA on substantial multithreaded applications and point out some of the information that it can provide to synthesis algorithms and designers.



**Figure 7: Portion of directed acyclic communication graph for ALPBench MPGenC.**

Figure 7 illustrates a portion of the directed acyclic communication graph produced by CETA for the MPGenC benchmark in the ALPBench benchmarks suite. Each node is labeled with its

corresponding temporal phase and PID. For example, 1.1040 is Step 1 for PID 1040. In total, we used 30 phases for the MPGenC phase partitioning, each of which was 1,000,000 CPU cycles long.

**Table 2: ALPBench Statistics Generated by CETA**

| Applications | Vertices | File size (KB) | CPU time (mins) | Simulation time (mins) |
|--------------|----------|----------------|-----------------|------------------------|
| MPGenc       | 16       | 4.9            | 0.077           | 20                     |
| MPGdec       | 13       | 2.8            | 0.009           | 3                      |
| Tachyon      | 27       | 150            | 2.725           | 50                     |
| Sphinx3      | 34       | 5600           | 1.406           | 851                    |
| FaceRec      | 3        | 380            | 35.073          | 5060                   |

In Figure 7, we show only 23 temporal phases due to space constraints. Compile-time parameters were chosen to allow MPGenc to use eight threads.

The graph in Figure 7 starts from PIDs 1039 and 1040: the main thread and the master thread. These threads communicate heavily with newly-created threads via shared memory. For example, PID 1050 is created in Step 8 and receives a high volume of data from the main thread, as well as other threads. During Step 9 the main thread receives no data because it is waiting for the other threads to complete local computations. In Step 11, after these local computations are finished, a large volume of data flows from other threads to the main and master threads. This results from the main and master threads reading the data computed by other threads.

CETA's main memory requirements are the result of the two Hash Map tables described in Section 3.3. CETA requires approximately eight bytes of memory for every unique physical address written during the simulation of an application. For MPGenc and Tachyon, the total memory required is approximately 246 MB and 2.87 GB, respectively.

In addition to communication patterns, CETA can also extract other information. Table 2 shows the number of vertices, the size of output file containing PID-to-PID dataflow information, the CPU time required from the viewpoint of the simulated application, and the actual time required for simulation for a number of benchmarks in the ALPBench suite. We used 1,000,000 CPU cycles as the temporal phase duration. The number of vertices is the number of the threads. This number is influenced by the ALPBench `NUM_THREADS` variable, which allows a user to specify the number of threads used by the benchmark applications. A `NUM_THREADS` value of eight was used for all benchmarks in Table 2. The sizes of the PID-to-PID dataflow files range from 2.8 KB to 306 KB. The time needed to transform output file into DOT format, and translate between graph formats, is less than two seconds for all applications. Simulation time is reported for CETA running on a 1 GHz AMD Sempron with 1 GB of RAM.

CETA is based on instruction-level simulation. This has some advantages, e.g., non-intrusiveness and no requirement for special-purpose hardware. However, there are also disadvantages. CETA faces the same simulation time problems as the host instruction-level simulator. When analyzing long-running applications, users of CETA can sample by enabling and disabling CETA during simulation using either magic instructions or Simics commands, or simply tolerate long simulation time for some applications.

In summary, CETA is capable of automatically extracting communication graphs from large multithreaded applications, e.g., MPGenc, MPGdec, Tachyon, and Sphinx3.

## 5. CONCLUSIONS

This article has presented CETA: a software package that automatically extracts communication graphs from threaded applications. CETA is composed of an instruction-level processor simulator module and a number of graph transformation scripts. It currently supports the analysis of applications for which either the source code or executables are available. The current release supports extraction of communication graphs from multithreaded Linux applications running on the Intel x86 architecture. CETA includes transformation routines to produce graphs suitable for a number of purposes, ranging from application optimization, to

multiprocessor synthesis, to developing thread assignment policies. It has been used to automatically extract runtime communication graphs from a number of multithreaded applications in the ALPBench benchmark suite. CETA is freely available for academic and non-profit use at <http://www.eecs.northwestern.edu/~dickrp/projects.html>.

## 6. REFERENCES

- [1] "AM". Finding out which process is running on top of an OS. Posting to Simics user's forum at <http://www.simics.net>.
- [2] CHEN, D.-K., SU, H.-M., AND YEW, P.-C. The impact of synchronization and granularity on parallel systems. In *Proc. Int. Symp. Computer Architecture* (May 1990), pp. 239–248.
- [3] COUMERI, S. L., AND THOMAS, D. E. A simulation environment for hardware-software codesign. In *Proc. Int. Conf. Computer Design* (Oct. 1995), pp. 58–63.
- [4] DICK, R. P., RHODES, D. L., AND WOLF, W. TGFF: Task graphs for free. In *Proc. Int. Wkshp. Hardware/Software Co-Design* (Mar. 1998), pp. 97–101.
- [5] GANSNER, E., KOUTSOFIOS, E., AND NORTH, S. Drawing graphs with dot. Tech. rep., AT&T Bell Labs, Feb. 2002.
- [6] GURUMURTHI, S., SIVASUBRAMANIAM, A., IRWIN, M. J., VIJAYKRISHNAN, N., KANDEMIR, M., LI, T., AND JOHN, L. K. Using complete machine simulation for software power estimation: The SoftWatt approach. In *Proc. Int. Symp. High Performance Computer Architecture* (Feb. 2002), pp. 141–150.
- [7] HU, J., AND MARCULESCU, R. Energy-aware mapping for tile-based NoC architectures under performance constraints. In *Proc. Asia & South Pacific Design Automation Conf.* (Mar. 2003), pp. 233–239.
- [8] KIANZAD, V., AND BHATTACHARYYA, S. S. CHARMED: A multi-objective co-synthesis framework for multi-mode embedded systems. In *Proc. Int. Conf. on Application-Specific Systems, Architectures and Processors* (Sept. 2004), pp. 28–40.
- [9] KIRSCHBAUM, A., BECKER, J., AND GLESNER, M. Run-time monitoring of communication activities in a rapid prototyping environment. In *Proc. Int. Wkshp. Rapid System Prototyping* (June 1998), pp. 52–57.
- [10] MAGNUSSON, P. S., DAHLGREN, F., GRAHN, H., KARLSSON, M., LARSSON, F., LUNDHOLM, F., MOESTEDT, A., NILSSON, J., STENSTRÖM, P., AND WERNER, B. SimICS/sun4m: A virtual workstation. In *Proc. USENIX Conf.* (June 1998).
- [11] PRAKASH, S., AND PARKER, A. SOS: Synthesis of application-specific heterogeneous multiprocessor systems. *J. Parallel & Distributed Computing* 16 (Dec. 1992), 338–351.
- [12] SASANKA, R., ADVE, S. V., CHEN, Y.-K., AND DEBES, E. The energy efficiency of CMP vs. SMT for multimedia workloads. In *Proc. Int. Conf. Supercomputing* (June 2004).
- [13] SCHMITZ, M. T., AL-HASHIMI, B. M., AND ELES, P. Iterative schedule optimization for voltage scalable distributed embedded systems. *ACM Trans. Embedded Computing Systems* 3, 1 (Feb. 2004), 182–217.
- [14] SIMUNIC, T., BENINI, L., AND DE MICHELI, G. Cycle-accurate simulation of energy consumption in embedded systems. In *Proc. Design Automation Conf.* (June 1999), pp. 867–872.
- [15] VALLERIO, K. S., AND JHA, N. K. Task graph transformation to aid system synthesis. In *Proc. Int. Conf. on Circuits & Systems* (May 2002), pp. 695–698.
- [16] VETTER, J. Dynamic statistical profiling of communication activity in distributed applications. In *Proc. Int. Conf. on Measurement and Modeling of Computer Systems* (June 2002), pp. 240–250.
- [17] XU, Z., R.LARUS, J., AND MILLER, B. P. Shared-memory performance profiling. In *Proc. Symp. on Principles and Practice of Parallel Programming* (June 1997), pp. 240–251.
- [18] YANG, P., WONG, C., MARCHAL, P., CATHOOR, F., DESMET, D., VERKEST, D., AND LAUWEREINS, R. Energy-aware runtime scheduling for embedded-multiprocessor SOCs. *IEEE Design & Test of Computers* 18, 5 (Sept. 2001).
- [19] YU, Y., AND PRASANNA, V. K. Energy-balanced task allocation for collaborative processing in networked embedded systems. In *Proc. Conf. on Language, Compiler and Tool Support for Embedded Systems* (June 2003).