Execution-Driven Tools for Parallel Simulation of Parallel Architectures and Applications (*)

David K. Poulsen and Pen-Chung Yew

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

EPG-sim is a newly-developed set of tools that performs execution-driven critical path simulation, trace generation, and simulation for serial, optimistically parallelized, and parallel application codes. These capabilities are integrated within a single framework through the use of intelligent source-level instrumentation. The ability to perform execution-driven simulations driven by optimistically parallelized codes, the ability to execute these simulations on parallel hosts, the use of source-level instrumentation, and the integration of the capabilities provided by EPG-sim are among the novel contributions of this work. EPG-sim has important uses in studying parallel architectures, parallelizing compilers, and parallel applications.

1. Introduction

Execution-driven simulation and tracing techniques have received considerable attention in recent years. Execution-driven techniques modify application codes by inserting additional *instrumentation* code. The resulting *instrumented codes* execute on a *host* machine using actual input data. Execution causes the generation of events reflecting the behavior of the original applications executing on a hypothetical *modeled* machine. Other event generation techniques include hardware monitoring, interrupt-based methods, pure simulation, and microcode-based methods [1]; these techniques can be used to study multiprogramming and operating system behavior in addition to application behavior. This paper concerns itself with execution-driven event generation techniques based on application code instrumentation, and tools resulting from the use of these techniques.

Execution-driven techniques have important uses in studying parallel systems. Parallel processor, memory system, and interconnect architectures can be modeled in order to study their parameters and the effect of varying these parameters on system or application performance. The effectiveness of parallelizing compilers, and of particular parallelizing transform techniques, can be evaluated. Dynamic profiling and analysis based on execution-driven techniques can be used to guide parallelizing compilers and to assess application parallelism and performance.

Execution-driven event generation may be used in *critical path simulation* (CPS), in *execution-driven trace generation* (ETG), to produce input for *trace-driven simulation* (TDS), or to drive *execution-driven simulation* (EDS). CPS directly executes instrumented serial codes such that they are implicitly, *optimistically parallelized* when executed on a uniprocessor host. This allows the potential parallelism and performance of these optimistically parallelized codes to be measured. ETG is merely the process of generating trace events via execution-driven techniques. TDS and EDS use these events to drive simulators, rather than using direct execution as in CPS.

The distinction between TDS and EDS is one of feedback. In TDS, no feedback (e.g., event timing or memory latency as determined by simulation) occurs that could affect subsequent event generation. In EDS, event generation and simulation are coupled so that feedback can occur. This feedback can be used to alter the ordering, timing, or latency of subsequently-generated events. In a simulated parallel code, the path of execution, and ordering and latency of events, may depend on the ordering and timing of preceding events [2]. In TDS, these dependences cannot be taken into account, limiting the accuracy of such simulations [3, 4]. EDS correctly models processor interactions. Another limitation of TDS is that traces, particularly those generated by non-execution-driven means, are biased to the particular system being traced, so they cannot be used arbitrarily in simulations of dissimilar systems (e.g., with different numbers of processors). EDS does not have

(*) This work was supported by the Department of Energy under grant No. DE FG02-85ER25001, the National Science Foundation under grant No. NSF MIP 89-20891, the National Security Agency, and an IBM Resident Study Fellowship.

such a limitation.

EPG-sim is a new set of instrumentation, tracing, and simulation tools that unifies and integrates execution-driven CPS, ETG, and EDS capabilities through the use of intelligent source-level instrumentation techniques. The tools have the ability to study serial, optimistically parallelized, or parallel codes, and the ability to model varying processor and system architectures. CPS, ETG, and EDS can be performed on uniprocessor or parallel hosts. EPG-sim provides a broader spectrum of capabilities than other tools described in the literature, since it supports all of these execution-driven techniques and provides good performance over a wide range of cost / accuracy tradeoffs. EPG-sim is robust, having the ability to simulate entire, full-size application codes using realistic input data, and is portable to different uniprocessor and parallel host machines.

This paper is organized as follows. First, background material on existing execution-driven systems is presented and EPG-sim is described in comparison to these systems. Next, the implementation of EPG-sim is described in detail. This includes information on source-level instrumentation, CPS, ETG, and EDS, and a brief discussion of accuracy issues. The last section of the paper presents experimental results that illustrate the functionality and usefulness of EPG-sim. A more extensive discussion of the systems, capabilities, and issues described in this paper can be found in [5].

2. Background

The design of EPG-sim addresses the need for an integrated set of tools that combines the best features of existing execution-driven simulation and tracing systems. The following paragraphs provide a brief survey of these systems and discuss the ways in which their capabilities are combined in EPG-sim.

Many execution-driven simulation and tracing systems have been described in the literature. MaxPar [6] is a CPS tool that measures the potential performance, parallelism, and behavior of optimistically parallelized codes given various architectural parameters. Other CPS tools have been used to measure program parallelism [7] and to perform dynamic dependence analysis [8]. ETG tools instrument serial codes for trace generation on uniprocessor hosts [1], or parallel codes for trace generation on parallel hosts [9, 10]. Examples of EDS systems include Tango [2], Proteus [11], and PEET [12]. These systems use serial or parallel codes to drive simulations that execute on uniprocessor hosts. Simulators exist that perform EDS for parallel codes on parallel hosts, but only for particular types of host architectures or modeled system architectures [13].

EPG-sim allows CPS, ETG, and EDS to be driven by serial, optimistically parallelized, or parallel codes. This can be done because of the extension of CPS instrumentation techniques to ETG and EDS. CPS executes on uniprocessor hosts and measures optimistically parallelized codes. ETG for serial codes executes on uniprocessor hosts, or for parallel codes on parallel hosts. In most cases, EDS for parallel codes executes on uniprocessor hosts. EPG-sim allows instrumented parallel codes to execute on uniprocessor or parallel hosts, and allows optimistically parallelized or parallel codes to drive parallel simulations.

Some CPS tools and most general EDS tools use machine-language-level instrumentation and event generation techniques. This results in processor models that are efficient but fairly architecture-specific. CPS tools that employ source-level instrumentation have used somewhat simplistic processor models. EPG-sim uses source-level instrumentation to implement a general, architecture-independent processor model that can model varying processor architectures at a reasonable level of detail. CPS is useful for low-cost, high-level architecture and application performance studies, but such simulators cannot model memory systems at a detailed level. EDS tools have the ability to model varying system architectures at varying levels of detail, limited only by the cost and complexity of the desired models. EPG-sim simultaneously supports modeling of varying processor and system architectures and combines CPS and EDS capabilities, resulting in a system that provides a wider range of cost / accuracy tradeoffs than other systems described in the literature.

3. Implementation

EPG-sim consists of a common set of instrumentation tools, a runtime library, and a set of parallel, discrete-event simulation tools (see Figure 1). The instrumentation tools use intelligent, source-level techniques and are implemented within a parallelizing compiler. The runtime library provides support for instrumented application codes and provides interfaces for tracing and simulation. The discrete-event simulation tools model varying parallel system architectures and implement EDS when coupled with instrumented codes. The following sections describe the various components of EPG-sim in more detail.

3.1 Source-level instrumentation

EPG-sim uses source-level instrumentation to generate events for CPS, ETG, and EDS. Source-level event generators are implemented by instrumenting application codes using tools based on the Parafrase-2 compiler [14],

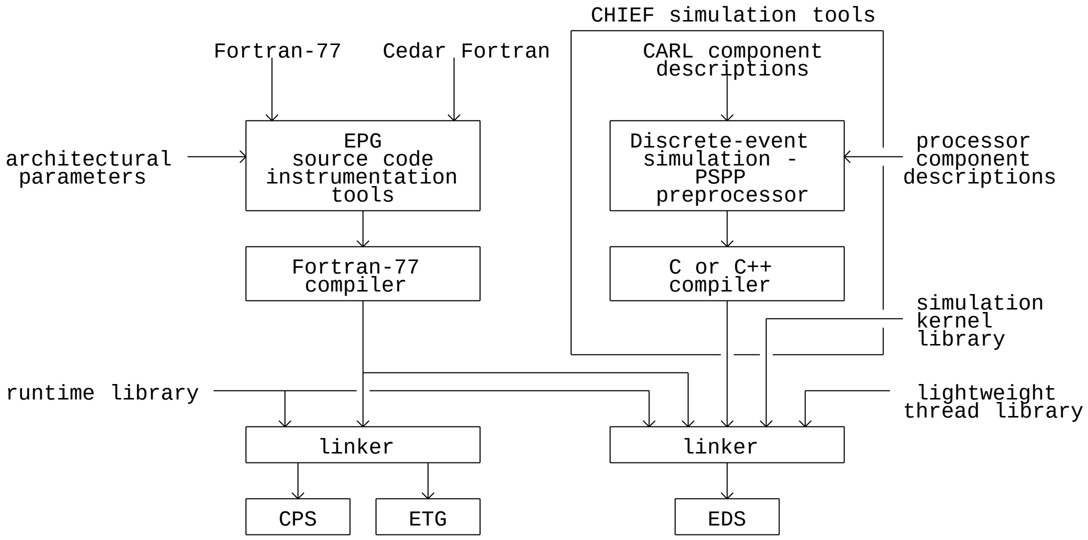

Figure 1 - EPG-sim Overview

developed at CSRD. Parafrase-2 supports Fortran-77 and Cedar Fortran [15], allowing both serial and parallel codes to be instrumented. The EPG (Execution-driven Program Generation) source code instrumentation tools are implemented as a set of passes within Parafrase-2. These tools perform instrumentation and constitute the front-end for EPG-sim. EPG handles source codes in a robust manner, with little or no user intervention, and produces efficient instrumented codes.

Events of interest are identified statically by EPG or by instrumenting to identify dynamic instances of events. Examples of events identified by EPG include: memory accesses, processor operations, task scheduling and dispatching operations, and static or dynamic data or control dependences. Dynamic dependence analysis is described in Section 3.2. Instrumentation is added for each event of interest to process that event at execution time. Examples of event processing include: generating traces, dispatching events to a simulator, histogramming or collecting statistics based on events, or performing direct simulation such as CPS (discussed in Section 3.2). Events may be associated with the value of simulation time at which they occur, or with the *static task* in the source code (a particular subroutine, loop, basic block, or statement) in which they occur. Histogramming allows profiles to be produced as a function of simulation time or of static tasks. Instrumentation may delay subsequent events in simulation time according to the occurrence of dynamic data or control dependences, feedback from EDS, or resource availability (e.g., processors, memory ports). The specific types of events identified and their corresponding instrumentation strategy varies according to the type of simulation being performed.

EPG uses a detailed, table-driven processor model that may be changed in order to model varying processor architectures. The table specifies the number of cycles of simulation time to assume for each individual arithmetic, logical, and assignment operation, and for overheads associated with variable access, array addressing, subprogram calls, loop startup, processor scheduling, and intertask or interprocedural memory access or synchronization. Fields in each table entry allow different values to be specified for scalar and pipelined vector operations. The processor model is flexible enough to describe varying architectures at a reasonably detailed level; however, implementation details like register and pipeline usage and conflicts are not modeled.

The processor model uses the notion of a task *grain size*, particularly when optimistically parallelizing codes. Task grain sizes include: subroutines, loop iterations (i.e., a DOALL / DOACROSS model of execution), basic blocks, statements, and individual operations. Tasks at the specified grain size are allowed to execute in parallel, completely or partially overlapping with the execution of other tasks, subject to their dynamic dependences or synchronization and resource availability. Operations within a particular task must execute serially. When performing optimistic parallelization, the number of simulated processors may either be *unconstrained*, allowing maximum parallelism in codes to be measured, or *fixed*, allowing the effects of processor scheduling policies to be observed. The processor scheduler is implemented as a runtime library routine that can be used to implement arbitrary scheduling policies. Scheduling strategies include round-robin, earliest available processor, and *near-optimal* [6], which attempts to maximize processor utilization.

3.2 Critical path simulation (CPS)

CPS in EPG-sim uses instrumentation to measure the minimum parallel execution time of optimistically parallelized codes, given various architectural and timing parameters. Variations on the techniques described here are also used to implement ETG and EDS, as discussed in the following sections. The techniques used are based on those used in MaxPar [6].

Minimum parallel execution time is measured by instrumenting to compute the earliest time at which each task could execute, given complete dynamic, interprocedural knowledge of data and control dependences, task grain size and associated serialization constraints, processor scheduling, resource constraints, and other parameters. Task completion times are determined by the timing of the serialized operations within them. An operation in a task may complete as soon as its dependences and serialization constraints are satisfied and the time for its operations and memory accesses has been assessed. Processor event timing is performed according to the model described in Section 3.1. In the most basic case, memory delays are also constant; more complex memory delay models are possible, as discussed below.

Dynamic data dependence analysis is accomplished by keeping track of times at which variables or array elements are available for reading or writing. These times are kept in *shadow variables* associated with each variable or array element. For a variable `X`, a *write shadow* `twX` stores the time at which `X` was last written, and a *read shadow* `trX` stores the time at which `X` was last read. If storage-related (anti and output) dependences are ignored, read shadows are not used. Dynamic interprocedural dependence analysis is implemented by passing appropriate shadows across subprogram call boundaries. To measure dynamic control dependences, `tCOND` shadows track times at which control dependences were satisfied.

Computations are performed using shadow variables to perform dynamic dependence analysis, determining the earliest time at which an operation could execute given its dynamic dependences. As an example, consider the addition operation `C = A + B`. For this operation, the instrumentation shown in Figure 2 is generated by EPG. The addition operation may begin as soon as anti and output dependences are satisfied for `C` (`twC`, `trC`), flow dependences are satisfied for `A` and `B` (`twA`, `twB`), and control dependences on the statement are satisfied (`tCOND`). The operation may complete as soon as its dependences are satisfied and the computation takes place (`tOPS`). Read shadows are updated according to the time the operation completes in the event that subsequent anti dependences involving `A` or `B` are encountered. The instrumentation for an operation is dense; however, good execution-time performance still results.

```
C = A + B
twC = max (twC,trC,twA,twB,tCOND)
      + tOPS
trA = max (trA,twC)
trB = max (trB,twC)
```

Figure 2 - CPS Instrumentation Example

Operations within a task are performed serially, so for task grain sizes consisting of multiple statements, an artificial flow dependence is introduced between sequential statements to serialize them. This is done using an additional shadow variable, `tTIME`, in the `max()` expression in the instrumentation for each statement. When the number of processors is fixed, a runtime library processor scheduling routine is called at the beginning of each task to select a processor according to a specified strategy. An additional shadow term, `tAVAIL`, is added to the `max()` expression for the first statement in each task to delay the start of that task until the scheduled processor is available.

To measure the effect of varying intertask memory delay or synchronization overhead, the distinction between intertask and intratask memory accesses must be made. To do this, tasks are uniquely numbered, and the current task number is kept in `tTASKid`. A *synchronization write shadow* `swX` is associated with each write shadow `twX`. Whenever `twX` is updated, `swX` is set to the `tTASKid` of the task that wrote `X`; this allows the instrumentation to determine at execution time if a variable to be read was written in the same task. Whenever `twX` is accessed, a delay is added to `twX` if `swX` does not match the current `tTASKid`; this assesses the intertask delay. Similarly, a *synchronization read shadow* `srX` is associated with `trX`, is updated whenever `X` is updated, and is checked whenever `trX` is accessed. To measure the effect of varying interprocedural synchronization overhead this same technique is used, only invocations of subprograms, not tasks, are uniquely numbered, and `tPROCid`, `pwX`, and `prX` shadows, corresponding to the `tTASKid`, `swX`, and `srX` shadows, are used.

CPS executes on uniprocessor hosts. To perform simulations using parallel Cedar Fortran codes as input, EPG converts these codes into serial Fortran-77 codes that still generate correct parallel timing information. This is done by converting parallel loops into serial loops and using shadow variables (e.g., `tTIME` shadows) to keep track of times at which parallel tasks *would have* executed if they had been executed in parallel. The resulting instrumented codes execute serially, but their shadow variables reflect their simulated parallel execution.

3.3 Trace generation (ETG)

ETG is accomplished in EPG-sim through the use of variations on the CPS instrumentation techniques described in Section 3.2. ETG is supported for serial, optimistically parallelized, and parallel codes. When performing ETG, events of interest are instrumented to generate trace information. Events of interest typically include memory references, or entry to or exit from particular static tasks. Trace information generated includes virtual address, data (from actual program execution), the value of simulation time, and processor number. Traces can also be generated in symbolic form, for instance, including variable names and subscript expressions for memory references. The following instrumentation is added for each memory reference event:

```
call genevent(opcode,addr,pid,time).
```

The exact content and form of the trace output is determined by `genevent`, a runtime library interface routine that is reconfigurable to suit particular applications.

ETG is performed for optimistically parallelized codes simply by using CPS instrumentation and adding `genevent` calls. For serial or parallel codes, the CPS instrumentation is modified to disable dynamic dependence analysis and optimistic parallelization (read and write shadows are no longer used), and to enforce the serialization required by the program semantics (using `tTIME` shadows, see Section 3.2). This results in only explicit parallelism being exploited. ETG for serial or optimistically parallelized codes executes on uniprocessor hosts. ETG for parallel codes may execute on uniprocessor or parallel hosts. When performing ETG for parallel codes on uniprocessor hosts, techniques similar to those used in CPS for parallel codes are used to convert parallel applications into instrumented serial codes.

3.4 Execution-driven simulation (EDS)

EDS in EPG-sim utilizes the ETG techniques described in Section 3.3 to form execution-driven event generators. These event generators are coupled with parallel discrete-event simulators to implement EDS. The overall structure of an EPG-sim EDS is shown in Figure 1.

CHIEF [16] parallel discrete-event simulators are used to construct system models and to perform simulation. The CHIEF system is portable, allowing simulations to be executed on a variety of uniprocessor and parallel host machines. A CHIEF simulator consists of a simulation model and a kernel library that drives the simulation. Multiple simulation paradigms are supported through the use of different kernel libraries. Paradigms currently supported use standard parallel discrete-event simulation, Chandy-Misra, or Time-Warp techniques [17]. Since CHIEF simulation kernels support parallel discrete-event simulation, EDS in EPG-sim is supported on uniprocessor or parallel hosts. Because of the features of the execution-driven event generators used, EDS in EPG-sim can be driven by serial, optimistically parallelized, or parallel codes. This results in the ability to perform parallel simulation of optimistically parallelized codes on parallel hosts.

CHIEF simulation models are described and constructed using a hardware description language, CARL [18]. CARL, an extension of C, supports hierarchical, component-based modeling of parallel systems at a wide range of levels of detail, from high-level abstract models to detailed behavioral register-transfer-level models. This allows models to be constructed at a level of complexity (and corresponding execution time cost) appropriate to the desired degree of simulation detail. Because of the hierarchical, component-based nature of CARL, these models can be used to construct other, more complex models. Processor models are special components, also described in CARL, that interface with execution-driven event generators. System models described in CARL are compiled for use with a particular kernel through a preprocessor, PSPP.

An EPG-sim EDS consists of a system model, processor models, instrumented application codes to drive the processor components, and runtime library interface routines. A thread of execution in an instrumented application code is associated with each processor component in the simulation model. Each thread, when executed, generates the sequence of simulation events produced by its processor. A distinct thread per processor can be provided by executing a different set of application code routines on each processor (i.e., a MIMD model of execution), or by executing a single application code in SPMD fashion, using one thread per processor. The latter approach is suitable for Cedar Fortran codes with parallel loops. In this case, instrumentation restructures loops so that they self-schedule iterations. For the parallel DOALL loop shown in Figure 3, the code is restructured so that multiple threads may execute it simultaneously.

The lower part of Figure 3 shows the resulting self-scheduling code (omitting any event instrumentation). Each thread gets a unique processor assignment from the `getvpn` call. The first iteration number (`inum`) and total number of iterations (`icnt`) for the loop are passed to the `loop_entry` routine, which stores these values so they can be shared by all threads. The `loop_entry` and `loop_exit` calls also perform barrier synchronizations that enforce the semantics of the parallel loop. Each `next_iter` call self-schedules a loop iteration, leaving the index value for that iteration in a private, per-thread copy of the loop index variable (`I`). Other program variables (`A`, `B`, `C`) are shared among threads.

```
doall I = M, N, P
      C(I) = A(I) + B(I)
end doall
      . . .
call getvpn(pid)
inum = M
icnt = max((N - M + P) / P, 0)
call loop_entry(inum,icnt)
while (next_iter(I,P)) do
      C(I) = A(I) + B(I)
end while
call loop_exit
```

Figure 3 - Self-Scheduled Parallel Loops

In order for a processor component to produce simulation events based on its thread in an application code, a lightweight thread library is used to switch between each processor component and its application thread. At any given point in simulation time, each processor component needs to know the next event to be issued by its thread. To determine this, the processor component performs a lightweight context switch into its thread and executes instrumented application code until an event is generated. Runtime library interface routines capture this event and perform a lightweight context switch back to the corresponding processor component. This mechanism ensures that events among multiple processors are issued and simulated in the correct order, even when a single application code is executed in SPMD fashion, or when simulating the execution of optimistically parallelized codes. The next time a processor component performs a lightweight context switch into its thread, the processor component returns the time at which the simulation of the previous event for that thread completed. This execution-driven feedback mechanism allows the timing of simulated events to affect the order and timing of subsequent events.

3.5 Accuracy

EPG-sim provides a wide range of cost / accuracy tradeoffs for simulating parallel systems by providing both CPS and EDS capabilities and by allowing varying processor and system architectures to simultaneously be simulated. The use of execution-driven techniques rather than trace-driven techniques increases the usefulness and accuracy of the tools. While the functionality of EPG-sim is the focus of this paper, accuracy issues are discussed briefly here.

CPS is appropriate for high-level architecture and application studies. Dynamic, interprocedural dependence analysis allows CPS to uncover parallelism not identified via static compiler analysis. Additional parallelism can be identified that might only have been uncovered through the use of parallelizing transformations. The major limitations of CPS - the ability to extract optimistic, but not optimal, parallelism, and the inability to model complex memory system architectures - are inherent in the techniques used, and are not limitations specific to EPG-sim. Parallelism measured using CPS is not optimal for two important reasons. First, CPS determines the critical path of execution through codes as presented; the effects of arbitrary code restructuring cannot be measured. Second, since CPS executes on uniprocessor hosts, tasks are dispatched to the processor scheduler in serial execution order. Although these tasks may execute in a different logical order than their physical dispatch order, the dispatch order can affect the optimality of the task schedule.

CPS cannot easily model memory latency effects caused by memory contention, network contention, or cache coherence activity. Constant memory delays are usually assumed; however, instrumentation can distinguish between intratask and intertask memory accesses and assign different delays accordingly. Runtime routines can be used to model more complex memory delays, but the interaction of memory accesses between processors is still difficult to model. EDS can be used to overcome these limitations, since it allows arbitrary system models to be constructed at any level of detail or abstraction desired.

One accuracy concern that remains in EPG-sim EDS is the accuracy of the processor models. Both CPS and EDS in EPG-sim use the architecture-independent processor model described in Section 3.1. The processor model is general and less architecture-specific than that of previously reported CPS or EDS systems. Processor models implemented using source-level instrumentation are not appropriate for modeling details of internal processor behavior such as internal register and pipeline usage and conflicts or instruction-sequence-specific timing. Register usage effects can be approximated through more sophisticated source-level instrumentation. For instance, EPG-sim can suppress events due to scalar variable accesses to model register usage.

4. Experimental results

The preceding sections describe the components, capabilities, and implementation of EPG-sim. This section presents some example results, illustrating the functionality and efficiency of the tools. Simulation results are used to demonstrate some of the event types that can be identified, to give simple examples of the kinds of data that can be derived, and to compare the performance of various types of simulations. The examples are intended only to illustrate the usefulness of EPG-sim; no detailed attempts are made to discuss the architectural significance of the results obtained.

CPS for several Perfect Benchmarks (R) codes [19] was performed to illustrate the usefulness and efficiency of such simulations. Most experiments were performed with unconstrained numbers of modeled processors (speedup results for fixed numbers of processors and near-optimal processor scheduling are also presented). Local (intratask) memory accesses, and arithmetic and logical operations, took one cycle each. Global (intertask) memory access delays took one cycle for most experiments, or were varied as indicated. Tasks were defined to be loop iterations, and storage-related data dependences were ignored. The dataset sizes of the codes studied were reduced in order to decrease execution time. These types of reductions do not change the relative performance of simulations when compared to native application speed.

code	parallel	speedup	s(512)
QCD	6065688	5.17	3.97
MDG	17068897	29.72	26.29
TRACK	4169294	28.33	24.09
BDNA	158069	1777.24	378.15
OCEAN	245670	312.73	192.60
DYFESM	381842	132.08	98.76
ARC2D	83108	1804.36	348.35
FLO52	91962	357.71	199.55
TRFD	11927	41.23	37.36
SPEC77	27801229	25.50	1.91

Figure 4 - Perfect Code Optimistic Parallelization

Figure 4 shows the minimum parallel execution time for optimistic parallelization of various Perfect codes using unconstrained numbers of processors. The maximum speedup (or *inherent parallelism* [6]) of each code, compared to serial execution time as calculated during CPS, is also given, followed by the speedup for 512 processors using near-optimal processor scheduling. The unconstrained processor speedups represent the maximum parallelism of the codes. Fixing the number of processors shows that some codes require large numbers of processors in order to exploit this parallelism effectively. Even with a fixed number of processors, CPS is able to identify considerably more parallelism than current parallelizing compilers [20].

Figure 5 is a histogram for the Perfect code TRACK displaying the number of task instantiations against parallel execution time. The data are sampled in large buckets; each point accumulates a task instantiation count for a range of approximately 20,000 cycles of simulation time. The profile clearly indicates two distinct phases of computation within the execution of the code; the first has more irregular parallelism, whereas the second exhibits a more regular pattern. Profiles of this type can be used to identify phases of computation that have unusually high or low parallelism, or can be plotted against static tasks to indicate sections of application codes with particular parallelism characteristics.

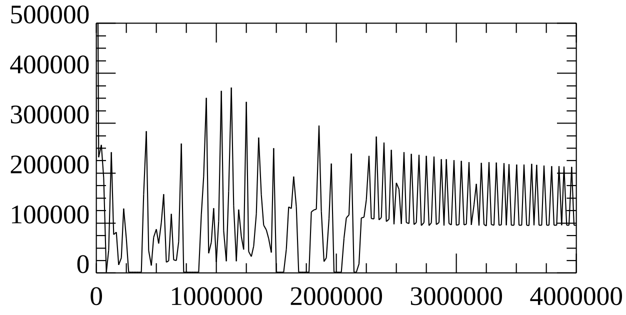

Figure 5 - TRACK Parallelism vs. Execution Time

Figure 6 shows speedup versus intertask communication overhead (global memory access delay) for various Perfect codes, normalized at zero overhead for each code to facilitate comparison. The decrease in speedup with increasing overhead indicates the relative sensitivity of each code's performance to this overhead. This decrease can also be interpreted as indicating the amount of intertask communication (or synchronization) in each code's critical path of execution. When parallel execution time, as opposed to speedup, is plotted against overhead (not shown), the increase in execution time is approximately linear in the amount of overhead; this explains the characteristic 1/x shape of the curves in Figure 6, and suggests that the effect of increased overhead is to lengthen the critical path of execution, not to cause other paths to become critical.

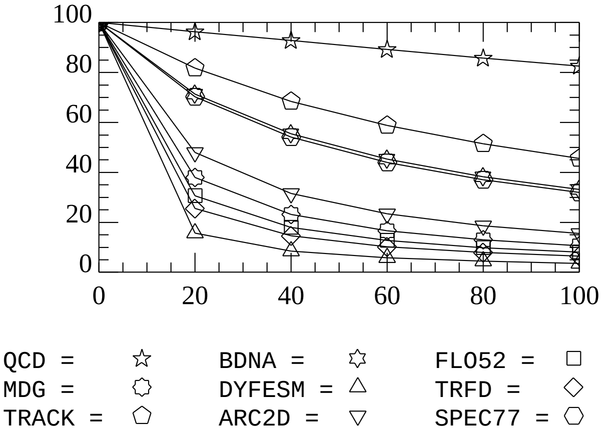

Figure 6 - Speedup vs. Communication Overhead

Additional experiments using CPS on the Perfect code TRFD were performed to compare various versions of this code. Experiments were performed with parameters and assumptions as in the previous CPS experiments. Three versions of TRFD were studied: TRFD, the original code; TRFD-p, a functionally equivalent parallel version of TRFD [21]; and TRFD-s, a serial version of TRFD-p.

code	num	speed	cpu
TRFD	unc	41.23	1.06
TRFD	512	37.36	65.31
TRFD	128	26.74	17.31
TRFD	32	15.28	5.31
TRFD	16	9.59	3.31
TRFD-s	unc	893.33	5.00
TRFD-s	512	185.06	325.00
TRFD-s	128	55.49	85.25
TRFD-s	32	18.34	26.00
TRFD-s	16	9.91	16.25
TRFD-p	unc	30.94	2.75
TRFD-p	512	30.94	3.50
TRFD-p	128	30.94	3.00
TRFD-p	32	22.61	3.00
TRFD-p	16	13.16	3.00

Figure 7 - TRFD Parallelization

Figure 7 shows the maximum speedup for optimistic parallelization of TRFD and TRFD-s, or for explicit parallelization of TRFD-p, using unconstrained or fixed numbers of processors. The time to execute each simulation is also shown, in user cpu seconds on a SUN SPARCstation-2 model 4/75. All execution times are normalized so that the raw execution time of each uninstrumented code is 1.0. The data for TRFD corresponds to the data in Figure 4. The results for TRFD-s show that restructuring reveals significant additional parallelism that could not be identified in TRFD; however, more processors are required to exploit this parallelism. Much of the additional parallelism is created by eliminating induction variables and scalar data dependences. The results for TRFD-p show the effect of exploiting only the explicit parallelism in TRFD-s. There is still significant inherent parallelism in the restructured code that is not explicit in parallel loops in TRFD-p.

The execution time results in Figure 7 show that CPS is efficient for unconstrained or small fixed numbers of processors. Execution time increases linearly with the (fixed) number of processors since near-optimal processor scheduling employs a linear search algorithm to locate the most suitable processor to schedule. Round-robin processor scheduling can be used to overcome this limitation while still obtaining reasonable schedules. Execution times for TRFD-p are lower because only explicit parallelism is being measured; this requires less instrumentation since dynamic dependence analysis is not performed.

TDS and EDS for TRFD-p were performed to compare their accuracy and performance. In either case, events were generated considering only the explicit parallelism in TRFD-p, and events for scalar memory references were suppressed to simulate register usage. The modeled machine used a dancehall architecture, uniform memory access delays, and processors with private caches connected by a crossbar. Experiments assumed 16 processors, one cycle processor to cache delays, one cycle cache hit delays, 100 cycle cache miss delays, and 100 cycle delays for cache invalidation messages. The cache coherence scheme used directories with pointer caches, unlimited processors per directory entry, and a simple three-state protocol. Memory and network contention were not modeled.

Figure 8 shows the results of various simulations of TRFD-p. Speedups compared to similar simulations with one processor are shown, along with average cache hit ratios and the time to execute each simulation (as in Figure 7). As expected, the speedups shown fall short of the ideal speedups predicted in Figure 7 (TRFD-p, 16 processors).

no.	speed	hit %	cpu
1	--	--	23.00
2	8.44	95.50%	53.25
3	8.44	95.50%	331.50
4	--	--	131.25
5	8.33	95.63%	378.75

Figure 8 - TRFD-p TDS and EDS

Line 1 in Figure 8 shows the time to perform EPG-sim ETG for TRFD-p assuming 16 processors. This time is longer than the time to perform the corresponding CPS (Figure 7, TRFD-p, 16 processors) due to the insertion of `genevent` calls at each memory reference. Lines 2 and 3 show the results for TDS using the traces generated in line 1. In line 2, the simulation used a straightforward model coded in C; in line 3, the EPG-sim EDS CHIEF model executed in TDS mode. As expected, lines 2 and 3 show the same simulation results, but varying execution times. Line 4 shows the time to perform the simulation in line 3 assuming all memory accesses are cache hits and with the cache components of the CHIEF model removed. The difference in execution times between lines 3 and 4 indicates the overhead of adding additional components to the CHIEF model.

Line 5 in Figure 8 shows the results of performing EPG-sim EDS for TRFD-p. This simulation produces slightly different results compared to TDS (line 3) due to the accuracy gained from allowing feedback during simulation. The differences are due primarily to differences in processor scheduling. TDS uses near-optimal processor scheduling, performed at ETG time, whereas EDS uses self-scheduling at execution time. While the difference in cache hit ratio is negligible, and the difference in speedup is only 1.3%, TRFD-p does not use any constructs that typically cause larger accuracy discrepancies [4].

The execution time required to generate events for EDS is the same as that required for ETG (line 1 in Figure 8) since the same instrumented code is executed in both cases. To make a fair comparison between execution times for TDS and EDS the time for ETG should be added to the TDS times. Considering this, the overhead of moving from EPG-sim TDS (line 1+3) to EDS (line 5) is not large. Most of the overhead incurred in moving from a straightforward C TDS (line 1+2) to an EPG-sim EDS is due to the introduction of CHIEF simulation models. This overhead is a strong function of the complexity of the CHIEF model, as illustrated by the difference in execution times between lines 3 and 4. While the overhead of using CHIEF models is large, using these models provides a wide range of cost / accuracy tradeoffs, allows simplified model construction and integration, and is necessary so that simulation events are correctly ordered.

CPS is a form of high-level execution-driven simulation that is useful in studying the parallelism of applications and the ability of architectures to exploit that parallelism. Basic CPS experiments, even with unconstrained numbers of processors, can produce useful performance information. CPS for entire Perfect codes using actual datasets is easily accomplished. Execution-time performance decreases by only three to five times (compared to native execution speed) when basic simulations for the Perfect codes are performed, despite the dense instrumentation required for these simulations. EDS incurs significant additional execution-time overhead, but allows system modeling at a considerably enhanced level of detail and accuracy. Most of this additional overhead is due to the use of CHIEF system models, while a smaller amount is due to the overhead of generating events to drive simulations. Some EDS performance is sacrificed in EPG-sim, when compared with other EDS systems (e.g., Proteus) in order to allow architecture-independent processor modeling and the integration of CPS.

5. Conclusions

Execution-driven techniques instrument application codes to generate events for simulation or tracing. These techniques are more accurate and versatile than trace-driven techniques and are preferable for studying parallel systems. Previously reported systems using execution-driven simulation and tracing techniques have numerous capabilities desirable in a general execution-driven system, including critical path simulation (CPS), trace generation (ETG), and simulation (EDS). CPS models the execution of optimistically parallelized codes on a hypothetical modeled system. This type of simulation reduces the dependence on parallel codes and hand parallelization when studying parallel architectures. EDS allows complex systems to be modeled in greater detail, with a correspondingly higher cost in execution time. The execution-driven capabilities described in the literature have been implemented using various source-level and machine-language-level approaches. The need exists for a general, flexible set of execution-driven tools, implemented within a single framework, that addresses all of these capabilities. Source-level instrumentation provides an attractive and functional way to accomplish this implementation and integration.

EPG-sim is a set of general-purpose execution-driven tools that performs parallel simulation and trace generation for studying parallel systems. These tools can model varying processor and system architectures, and can simulate the effects of serial, optimistically parallelized, or parallel codes being executed on modeled parallel systems. CPS, ETG, and EDS can be performed, and simulations can be executed on uniprocessor or parallel hosts. These capabilities are integrated into a single execution-driven system through the use of source-level instrumentation. EPG-sim is portable to different uniprocessor and parallel host machines, increasing the flexibility of the system. The ability to perform EDS with optimistically parallelized codes driving modeled parallel systems, the ability to execute these simulations on parallel hosts, the use of source-level instrumentation, and the integration and breadth of capabilities provided by EPG-sim are among the novel contributions of this work.

Source-level instrumentation provides many useful benefits: it allows the integration of the various capabilities of EPG-sim; and it provides intelligent and efficient instrumentation that is architecture independent, is tailored to specific types of simulation, and provides good execution-time performance. These features result in a system providing an exceptionally broad range of cost / accuracy tradeoffs, supporting the notion of paying in simulation complexity only for the level of detail required. The use of EPG to implement CPS has produced several useful features compared to other CPS systems. EPG instrumentation is robust and expandable, and it offers good performance, both in terms of performing instrumentation and in terms of simulation cost. EPG instrumentation models processor architectures in a detailed and less architecture-specific manner, allowing the effects of varying architectural and timing parameters on performance to be studied in greater detail. The ability to measure interprocedural synchronization overhead is a useful extension of previous CPS work.

Acknowledgments

The design of EPG-sim owes much to several other projects and individuals at CSRD, notably: Parafrase-2 (Bruce Leung, Mohammad Haghighat, and Dale Schouten), MaxPar (Ding-Kai Chen), CPAT (Paul Petersen), and CHIEF (Pavlos Konas). The authors also wish to thank Dr. Josep Torrellas, John Andrews, and Carl Beckmann for their interest in this paper and the tools it describes, and Merle Levy and Angeline Gliniecki for detailed, last-minute proofreading.

References

1. Stunkel, C. B., Janssens, B., and Fuchs, W. K., "Address Tracing for Parallel Machines", IEEE Computer, January 1991, pp. 31-38.
2. Davis, H., Goldschmidt, S., and Hennessy, J., "Multiprocessor Simulation and Tracing Using Tango", Proceedings of ICPP 1991, pp. II-99-106.
3. Bitar, P., "A Critique of Trace-Driven Simulation for Shared-Memory Multiprocessors", in Dubois, M., and Thakkar, S. (eds.), 16th ISCA Workshop on Cache and Interconnect Architectures in Multiprocessors, Boston: Kluwer Academic, 1989.
4. Goldschmidt, S. R., and Hennessy, J. L., "The Accuracy of Trace-Driven Simulations of Multiprocessors", Proceedings of SIGMETRICS 1993, pp. 146-157.
5. Poulsen, D. K., and Yew, P.-C., "Execution-Driven Tools for Parallel Simulation of Parallel Architectures and Applications", Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, CSRD Report No. 1280, September 1993.
6. Chen, D.-K., Su, H.-M., and Yew, P.-C., "The Impact of Synchronization and Granularity on Parallel Systems", Proceedings of ISCA 1990, pp. 239-249.
7. Larus, J., "Parallelism in Numeric and Symbolic Programs", Third Workshop on Programming Languages and Compilers for Parallel Computing, August 1990.
8. Petersen, P., and Padua, D., "Dynamic Dependence Analysis: A Novel Method for Data Dependence Evaluation", Presented at the 5th Annual Workshop on Languages and Compilers for Parallel Computing, New Haven, CT, August 3-5, 1992.
9. Eggers, S., Keppel, D., Koldinger, E., and Levy, H., "Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor", Proceedings of SIGMETRICS 1990, pp. 37-47.
10. Stunkel, C. B., and Fuchs, W. K., "TRAPEDS: Producing Traces for Multicomputers via Execution-Driven Simulation", Proceedings of SIGMETRICS 1989, pp. 70-78.
11. Brewer, E., Dellarocas, C., Colbrook, A., and Weihl, W., "Proteus: A High Performance Parallel Architecture Simulator", Laboratory for Computer Science, MIT, MIT/LCS/TR-516, September 1991.
12. Grunwald, D., Nutt, G. J., Sloane, A. M., Wagner, D., and Zorn, B., "A Testbed for Studying Parallel Programs and Parallel Execution Architectures", Technical Report, Department of Computer Science, University of Colorado, Boulder, April 28, 1992.
13. Reinhardt, S. K., Hill, M. D., Larus, J. R., Lebeck, A. R., Lewis, J. C., and Wood, D. A., "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers", Proceedings of SIGMETRICS 1993, pp. 48-60.
14. Polychronopoulos, C. D., Girkar, M. B., Haghighat, M. R., Lee, C. L., Leung, B. P., and Schouten, D. A., "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors", Proceedings of ICPP 1989, pp. II-39-48.
15. Hoeflinger, J., "Cedar Fortran Programmer’s Handbook", Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, CSRD Report No. 1157, October 1991.
16. Bruner, J., Cheong, H., Veidenbaum, A., and Yew, P.-C., "CHIEF: A Parallel Simulation Environment for Parallel Systems", Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, CSRD Report No. 1050, November 1990.
17. Konas, P., and Yew, P.-C., "Synchronous Parallel Discrete-Event Simulation on Shared-Memory Multiprocessors", Presented at the 6th Workshop on Parallel and Distributed Simulations, Newport Beach, CA, January 1992.
18. Beckmann, C. J., "CARL: An Architecture Simulation Language", Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, CSRD Report No. 1066, December 1990.
19. Berry, M., Chen, D.-K., Koss, P., Kuck, D., Pointer, L., Lo, S., Pang, Y., Roloff, R., Sameh, A., Clementi, E., Chin, S., Schneider, D., Fox, G., Messina, P., Walker, D., Hsiung, C., Schwarzmeier, J., Lue, K., Orszag, S., Seidl, F., Johnson, O., Swanson, G., Goodrum, R., and Martin, J., "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers", International Journal of Supercomputer Applications, Vol. 3, No. 3, Fall 1989, pp. 5-40.
20. Blume, W., and Eigenmann, R., "Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks (R) Programs", IEEE Transactions on Parallel and Distributed Systems, Vol. 3, No. 6, November 1992, pp. 643-656.
21. Andrews, J. A., and Gallivan, K. A., "Analysis of a Cedar Implementation of TRFD", Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, CSRD Report No. 1312, August 1993.