

An Architecture Independent Approach to Emulating Computation Intensive Workload for Early Integration Testing of Enterprise DRE Systems

James H. Hill

Department of Computer and Information Science
Indiana University/Purdue University at Indianapolis
Indianapolis, IN, USA
hillj@cs.iupui.edu

Abstract. Enterprise distributed real-time and embedded (DRE) systems are increasingly using high-performance computing architectures, such as dual-core architectures, multi-core architectures, and parallel computing architectures, to achieve optimal performance. Performing system integration tests on such architectures in realistic operating environments during early phases of the software lifecycle, *i.e.*, before complete system integration time, is becoming more critical. This helps distributed system developers and testers evaluate and locate potential performance bottlenecks before they become too costly to locate and rectify. Traditional approaches either (1) rely heavily on simulation techniques or (2) are too low-level and fall outside the domain knowledge distributed system developers and testers. Consequently, it is hard for distributed system developers and testers to produce realistic operating conditions for early integration testing of such systems.

This paper provides two contributions to facilitating early system integration testing of enterprise DRE systems. First, it provides a generalized technique for emulating computation intensive workload irrespective of the target architecture. Secondly, this paper illustrates how the emulation technique is used to evaluating different high-performance computing architectures in early phases of the software lifecycle. The technique presented in this paper is empirically and quantitatively evaluated in the context of a representative enterprise DRE system from the domain of shipboard computing environments.

1 Introduction

Emerging trends in enterprise distributed real-time and embedded systems. Enterprise distributed real-time and embedded (DRE) systems, such as mission avionics systems, shipboard computing environments, and traffic management systems, are increasingly using high-performance computing architectures [12, 17], *e.g.*, multi-threaded, hyper-threaded, and multi-core processors. High-performance computing architectures help increase parallelization of computation intensive workload, which in turn can improve overall performance of enterprise DRE systems [17]. Furthermore, such computing architectures enable enterprise DRE systems to scale to the computation needs of next generation enterprise DRE systems, such as ultra-large-scale systems [13].

As enterprise DRE systems grow in both complexity and scale, it is becoming more critical to evaluate the system under development on different high-performance computing architectures early in the software lifecycle, *i.e.*, before complete system integration time. This enables distributed system developers to determine which architecture is best for their needs. It also helps distributed system developers avoid the *serialized-phasing development problem* [19] where infrastructure- and application-level system entities are developed and validated in different phases of the software lifecycle, but fail to meet performance requirements when integrated and deployed together on the target architecture. Serialized-phasing development therefore makes it hard for distributed system developers and testers to identify potential performance bottlenecks before they become too costly to locate and rectify.

Existing techniques for evaluating and validating computation intensive systems on high-performance computing architectures early in the software lifecycle to overcome the serialized-phasing development problem rely heavily on simulation and/or analytical models [2–6]. Although this approach is feasible for predicting performance in early phases of the software lifecycle [18], such techniques cannot account for all the complexities of enterprise DRE systems (*e.g.*, the operating environment and underlying middleware). These complexities and others, such as arrival rates of events and thread-/lock synchronization, are known to affect computation intensive workload and overall performance of such systems. Distributed system developers therefore need improved techniques for evaluating enterprise DRE systems on different high-performance computing architectures in early phases of the software lifecycle.

Solution approach → **Abstracting computation intensive workload via emulation techniques.** Emulation [18, 20] is an approach for constructing operational environments that are capable of generating realistic results. In the context of high-performance computing architectures, emulating computation intensive workload on the target architecture and operational environment enables distributed system developers and testers to construct realistic operational scenarios for evaluating different high-performance computing architectures. Moreover, it allows distributed system developers and testers to evaluate and validate system performance, such as latency, response time, and service time, when complexities of enterprise DRE systems are taken into account.

This paper describes a technique for emulating computation intensive workload to evaluate different high-performance computing architectures and evaluate and validate enterprise DRE system performance early in phases of the software lifecycle. The emulation technique presented in this paper abstracts away optimizations of high-performance computing architectures, such as caching and look-ahead processing, while letting the target architecture handle execution concerns, such as context-switching and parallel processing, which can vary between different high-performance architectures. The emulation technique is able to accurately emulate computation intensive workloads ranging from [1, 1000] msec irrespective of the underlying high-performance computing architecture. Experience gained from applying the emulation technique presented in this paper shows that it raises the level of abstraction for performance testing so that distributed system developers and testers focus more on locating potential performance

bottlenecks instead of wrestling with low-level architectural concerns when constructing realistic operational environments.

Paper organization. The remainder of this paper is organized as follows: Section 2 introduces a case study for a representative enterprise DRE system; Section 3 presents the technique for emulating computation intensive workload independent of the underlying high-performance computing architecture; Section 4 presents empirical results for the emulation technique in the context of the case study; Section 5 compares this work with other related work; and Section 6 provides concluding remarks.

2 Case Study: The SLICE Scenario

The SLICE scenario is an representative enterprise DRE system from the domain of shipboard computing environments. It has been used in prior research and multiple case studies, such as evaluating system execution modeling tools [8, 11], conducting formal verification [9], and highlighting challenges of searching the deployment and configuration solution space of such systems [10]. Figure 1 shows an high-level model of the SLICE scenario.

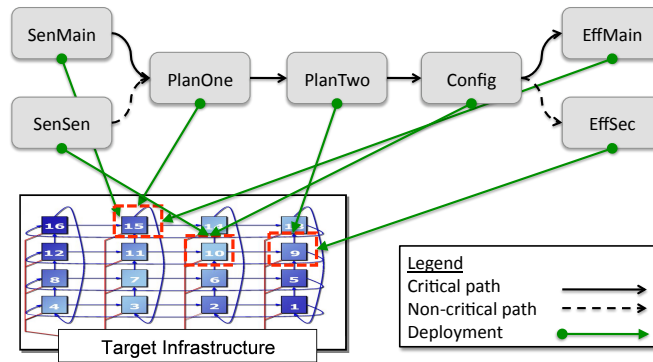


Fig. 1. High-level overview of the SLICE scenario.

To briefly reiterate an overview of the SLICE scenario, Figure 1 illustrates that the SLICE scenario is composed of seven different component¹ instances named: (from left to right): *SenMain*, *SenSec*, *PlanOne*, *PlanTwo*, *Config*, *EffMain*, and *EffSec*. The directed lines between each component represents inter-communication between components, *e.g.*, sending/receiving an event. The SLICE scenario also has a *critical path* of execution, which is represented by the solid directed lines between components, that must be executed in a timely manner. Finally, each of the components in the SLICE scenario must be deployed (or placed) on one of three hosts in the target environment.

¹ In context of the SLICE scenario, a *component* is defined an abstraction that encapsulates common services that can be reused across different application domains.

The SLICE scenario is an application-level entity, however, the infrastructure-level entities it will leverage are currently under development. The SLICE scenario is therefore affected by the serialized-phasing development problem discussed in Section 1. To overcome the effects of serialized-phasing development, system developers are using system execution modeling tools to conduct system integration test for the SLICE scenario, such as evaluating end-to-end response time of its critical path, during early phases of the software lifecycle, *i.e.*, before complete system integration.

In particular, system developers are using the CUTS [11] system execution modeling tool, which is designed for component-based distributed systems, to conduct system integration tests during early phases of the software lifecycle, and to overcome the serialized-phasing development problem. CUTS uses domain-specific modeling languages [16] and emulation techniques to enable system integration testing during early phases of the software lifecycle on the target architecture using components that look and feel like their counterparts under development. Likewise, as the *real* components are developed, they can seamlessly replace the *faux* components to facilitate continuous system integration testing throughout the software lifecycle.

Similar to prior work [11], system developers intended to use CUTS to evaluate the end-to-end response time of the SLICE scenario's critical path. This time, however, system developers plan to extend their previous testing efforts and evaluate the end-to-end response time of the SLICE scenario's critical path on different high-performance computing architectures, such as multi-threaded, hyper-threaded, and multi-core architectures. Applying CUTS to evaluate the SLICE scenario on different high-performance computing architectures, however, revealed the following limitations:

- **Limitation 1. Inability to easily adapt to different high-performance computing architectures.** Different high-performance computing architectures have different hardware specifications, such as processor speed and number of processors. Different hardware specifications also affect the behavior of computation intensive workload. For example, execution time for two separate threads on a dual-core architecture will be less than execution time for the same threads on a single processor architecture—assuming there are no blocking affects, such as waiting for a lock, between the two separate threads of execution.

System developers can use the high-performance computing architecture's hardware specification to model the behavior of computation intensive workload for emulation purposes. This approach, however, is too low-level and outside their knowledge domain. Likewise, such a model can be too costly to construct and validate, and can negatively impact overall emulation performance on the target architecture. System developers therefore need a technique that can easily adapt to different high-performance computing architecture and provide accurate emulation of CPU workload.

- **Limitation 2. Inability to adapt and scale to large number of computation resources.** System developers plan to leverage dynamic testbeds, such as Emulab [21], to conduct their integration tests. Emulab enables system developers to configure different topologies and operating systems to produce a realistic target environment for distributed system integration testing.

Although the SLICE scenario consists of three separate host, system developers intend to conduct scalability tests by increasing both the number of components and hosts in the SLICE scenario. System developers therefore need lightweight techniques that will enable them to rapidly include additional resources in their experiments, and still provide accurate emulation of computation intensive workload.

Due to these limitations it is hard for system developers of the SLICE scenario to evaluate the end-to-end response time of its critical path on different high-performance computing architecture. Moreover, this problem extends beyond the SLICE scenario and applies to other distributed systems that need to evaluate system performance on different (high-performance) computing architectures. The remainder of this paper, therefore, discusses a technique for overcoming the aforementioned limitations and improving CUTS to enable early system integration testing of computation intensive enterprise DRE systems on different high-performance computing architectures.

3 Architecture Independent Approach for Accurate Emulation of CPU Workload

This section discusses a technique for accurately emulating computation intensive workload independent of the underlying high-performance computing architecture. The approach abstracts CPU workload and focuses on overall execution time of the computation intensive workload under emulation.

3.1 Abstracting CPU Workload for Emulation

Conducting system integration test (1) during early stages of the software lifecycle, (2) on the target architecture, and (3) in the target environment enables distributed system developers to obtain realistic feedback for the system under development. Moreover, it enables distributed system developers to locate potential performance bottlenecks so they can be rectified in a timely manner. The ability to locate such performance bottlenecks, however, depends heavily on the accuracy of such tests, *i.e.*, its behavior and workload, conducted during early phases of the software lifecycle.

In the context of high-performance computing architectures, it is possible to construct fine-grained models that will accurately emulate effects of their characteristics, such as enhanced performance due to CPU caching or look-ahead processing, or fetching instructions from memory. For example, Figure 2 illustrates an emulation/execution model for (a) fetching instructions from memory and (2) CPU caching effects. As illustrated in Figure 2, in the case of (a) fetching instructions from memory, a portion of the computation intensive workload (*i.e.*, overall execution time) is attributed to fetching the instructions from memory. Likewise, in the case of (b) CPU caching effects, the portion of the overall execution time that would have been attributed to fetching CPU instructions from memory no longer exists.

Because high-performance computing architectures have many characteristics that can impact performance, it is not feasible to construct fine-grained emulation models that capture all effects—especially when conducting system integration tests during

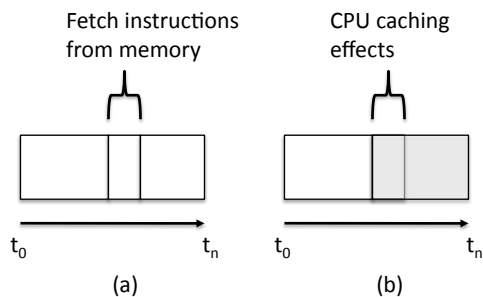


Fig. 2. Emulation/execution model to high-performance computing effects.

early phases of the software lifecycle. Instead, a more feasible approach is abstracting away such effects and focusing primarily on overall execution time, similar to profiling [7]. This is possible because as the same computation (re)occurs many times throughout the lifetime of a system, it will converge on an average execution time. The average execution time will therefore incorporate the effects from characteristics of its underlying high-performance computing architecture.

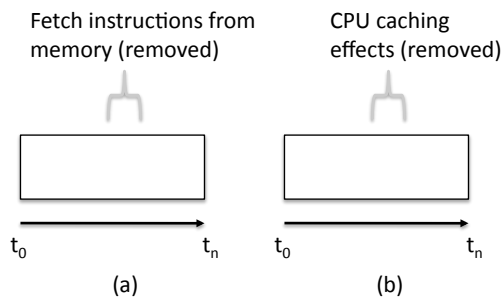


Fig. 3. Example of abstracting the emulation/execution model for computing effects.

Figure 3 highlights abstracting the effects presented in Figure 2. As shown in Figure 3, instead of modeling each individual effect, the average execution time for the computation intensive workload is modeled. For example, (a) does not explicitly model the effects of fetching instructions from memory. Instead (a) models the average execution time of the computation intensive workload, which includes the execution time for occasionally fetching instructions from memory. The same holds to be true for (b), *i.e.*, instead of modeling performance gain from CPU caching effects, the average execution time for the computation intensive workload is modeled, which includes such effects. The remainder of this section discusses how abstracting CPU effects is realized when emulating computation intensive workload for enterprise DRE systems.

3.2 Realizing Abstraction of Computation Intensive Workload in CUTS

In Section 3.1, a technique for abstracting the different CPU effects of high-performance computing architectures was discussed. The main thrust of the technique focuses on overall average execution time of a computation intensive workload (or emulated CPU workload), instead of modeling each individual CPU effect—especially when conducting system integration tests during early phases of the software lifecycle. This, in turn, simplifies emulating computation intensive workload for different high-performance computing architectures.

Since overall average execution time is the main focus of the abstraction technique, it is therefore feasible to use an arbitrary computation to represent emulated CPU workload. The main challenge, however, is ensuring the arbitrary computation is capable of accurately representing different average execution times, *i.e.*, scaling to different CPU (or computation) workloads. In CUTS (see Section 2), this challenge is resolved by using a calibration factor for an arbitrary CPU workload. The calibration factor is then used to scale the arbitrary CPU workload to different computational needs, *i.e.*, different average execution times. Algorithm 1 presents the algorithm for calibrating the CPU workload generator that realizes the abstraction technique in CUTS.

As illustrated in Algorithm 1, the goal of the calibration effort is to derive a calibration factor *calib* that will achieve the scaling factor f (or time). As highlighted in Algorithm 1, the initial bounds of the calibration factor is defined as $[0, MAX_INT]$. For each iteration at deriving the correct calibration factor for the CPU workload (or abstract computation), the median value is used as the potential calibration factor. If the calibration factor yields an execution time above the target scaling factor, then the current calibration factor becomes the upper bound. Likewise, if the calibration factor yields an execution time below the target scaling factor, then the current calibration factor becomes the lower bound.

This process continues until either (1) the calibration factor yields an execution time that is within acceptable error of the target scaling factor or (2) the lower and upper bounds are equal. In the case that the lower and upper bounds are equal and the calibration factor is not derived, the calibration effort is tried for up to *max* times. This can occur if there is background noise during the calibration exercise. If a calibration factor is derived, then it is verified that it will accurately generate execution times up to a user-defined execution time by scaling the calibration factor based on the number of iterations need to reach a target execution time. Finally, if the verification process fails, then the average error of the verification process for different execution times is used to adjust the current calibration factor for the next attempt.

By using the calibration exercise presented in Algorithm 1 it is possible to accurately emulate CPU workload independent of the underlying computational resources on different high-performance computing architectures, which has been realized in CUTS. Moreover, it enables emulation of computation intensive workload based on CPU time and (1) not “wall time” or (2) having to monitor how much time a given thread has currently executed on the CPU.

Algorithm 1 General algorithm for calibrating CPU workload generator that abstracts CPU effects.

```
1: procedure CALIBRATE( $f, \delta, \max$ )
2:    $f$ : target scaling factor for workload
3:    $\delta$ : acceptable error in calibration
4:    $\max$ : maximum number of attempts for calibration
5:    $i \leftarrow 0$ 
6:    $\text{bounds} \leftarrow \{0, \text{MAX\_INT}\}$ 
7:    $\text{calib} \leftarrow 0$ 
8:
9:   while  $i < \max$  do
10:    while  $\text{bounds}[0] \neq \text{bounds}[1]$  do
11:       $\text{calib} \leftarrow (\text{bounds}[0] + \text{bounds}[1])/2$ 
12:       $t \leftarrow \text{exec}(\text{computation}, \text{calib})$ 
13:
14:      if  $t > f + \delta$  then
15:         $\text{bounds}[1] \leftarrow \text{calib}$ 
16:      else if  $t < f - \delta$  then
17:         $\text{bounds}[0] \leftarrow \text{calib}$ 
18:      else
19:        break
20:      end if
21:    end while
22:
23:     $\text{done} \leftarrow \text{VERIFY}(\text{calib})$ 
24:    if  $\text{done} = \text{true}$  then
25:      return  $\text{calib}$ 
26:    end if
27:  end while
28: end procedure
```

4 Evaluating Abstraction Technique for Emulating Computation Intensive Workload

The section presents the results for validating the computation intensive workload generator discussed in Section 3. This section also presents the results for applying the computation intensive workload generator to the SLICE scenario introduced in Section 2. It is necessary to validate the workload generator because it will ensure that system developers of the SLICE scenario are able to accurately emulate CPU workload for their early integration tests. Moreover, as components are collocated (*i.e.*, placed on the same host), the average execution time of its computation intensive workload is expected to be longer due to software/hardware contention [18, 20] than when the same components are deployed in isolation. It is therefore necessary to validate that the abstraction technique can produce such behavior/results.

The experiments described below (unless mentioned otherwise) were run in a representative target environment testbed at ISISlab (www.isislab.vanderbilt.edu). ISISlab is powered by Emulab software that configures network topologies and operating systems to produce a realistic target environment for distributed system integration testing. Each host in the experiment was an IBM Blade type L20, dual-CPU 2.8 GHz processor with 1 GB RAM configured with the Fedora Core 6 operating system. Figure 4 shows a representative illustration of ISISlab at Vanderbilt University.

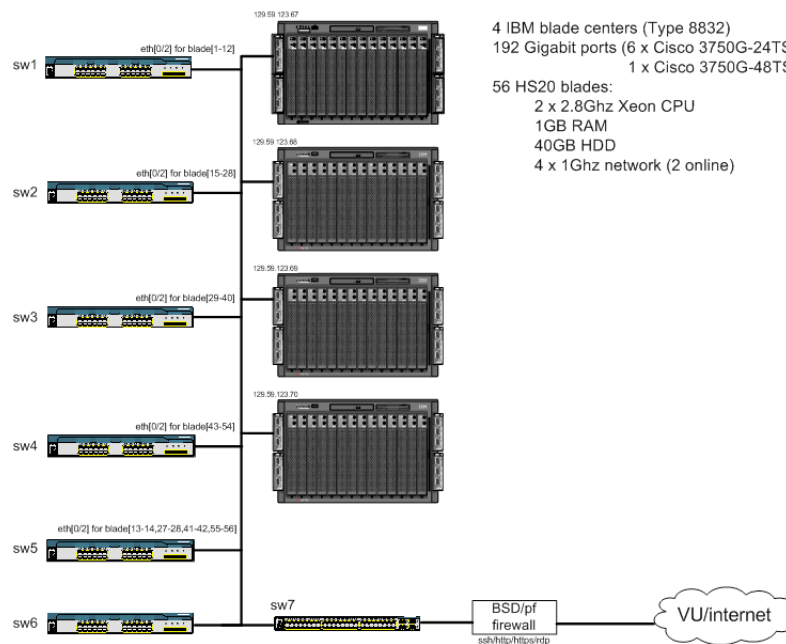


Fig. 4. ISISlab at Vanderbilt University

4.1 Validating the Calibration and Emulation Technique

Determining the upper bound of the emulation technique discussed in Section 3.2 will enable distributed system developers to understand how much CPU workload can accurately be guaranteed before the emulation becomes unstable. When calibrating the CPU workload generator using Algorithm 1 in Section 3.2 to determine this upper bound, all tests were conducted in an isolated environment on the target host. This prevents any interference from other process that may be executing in parallel on the target host.² Figure 5 highlights results of the calibration exercise when the scaling factor f is 20000 usec, acceptable error δ is 100 usec, and max is 10.

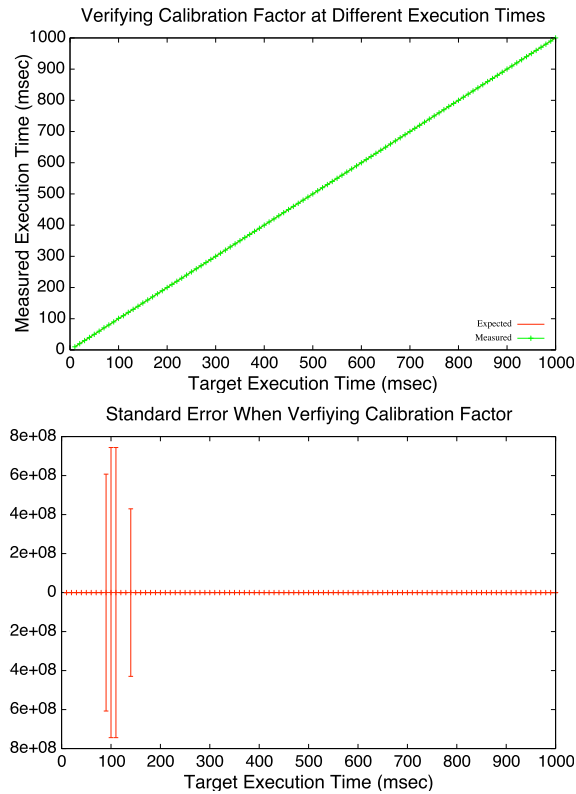


Fig. 5. Calibration results for the CUTS CPU workload generator.

As illustrated in Figure 5, once the CPU workload generator is calibrated using Algorithm 1, the emulation technique is able to accurately generate computation inten-

² It is hard to produce a completely isolated environment because a host may have background services running. It is assumed that such services, however, are negligible and do not interfere with the calibration exercise.

sive workloads up to 1000 msec (or 1 second) as illustrated in the upper graph. The lower graph illustrates the standard error for each of the execution times during verification process where the most error occurred between [70, 150] msec. After 1000 msec of computation, the emulation becomes unstable is not able to accurately emulate the computation intensive workload (not shown in Figure 5). It is believed that the inaccuracy after 1000 msec is attributed to the fact that is hard to guarantee long running processes will occupy the CPU without real-time scheduling capabilities.

The calibration and verification results in presented in Figure 5 took only 1 try. This means that distributed system developers that who want to evaluate system performance have to ensure their CPU workload is less than 1000 msec, which is well above the upper bound for many short running computations of enterprise DRE systems, such as in the SLICE scenario. More importantly, they have a simple technique that will accurately emulate computation intensive workload without modeling low-level architecture concerns (*i.e.*, addresses Limitation 1 in Section 2).

Validating the calibration technique on multiple homogeneous hosts. The emulation technique presented in Section 3.2 and validated above enables system developers to accurately emulate computation intensive workload up to 1000 msec. This emulation technique, however, works only on the host on which the calibration was performed. Distributed system developers intend to use ISISlab to conduct early system integration tests of the SLICE scenario. Executing calibration test of each of the hosts in ISISlab, however, is hard and time-consuming because it requires isolated access to each host in a resource-sharing environment.

Distributed system developers of the SLICE scenario therefore hypothesize that it is possible to use the same calibration for different homogeneous hosts, or architectures. If this hypothesis is true, then it will reduce the complexity of managing and configuring integration testbeds. For example, if distributed system developers elect to use integration testbeds like ISISlab, then they have to only calibrate the CPU workload generator once on each class of hosts.

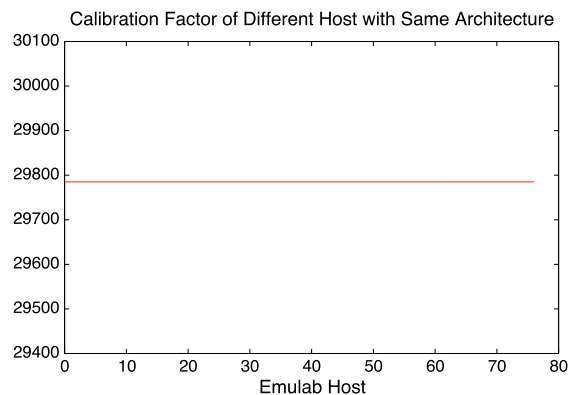


Fig. 6. Validating calibration technique on different hosts with same architecture.

Figure 6 illustrates the results for calibrating the CPU workload generator on 77 different hosts in Emulab (www.emulab.com). Emulab was used for this experiment because it has more hosts for testing than ISISlab. Each host in Emulab used for this experiment was a Dell PowerEdge 2850s with a single 3 GHz processor, 2 GB of RAM, 2 x 10,000 RPM 146 GB SCSI disks configured with a Linux 2.6.19 Uniprocessor kernel. As illustrated in Figure 6, each host in the experiment yielded the same calibration factor. The distributed system developer’s hypothesis was therefore true (*i.e.*, addresses Limitation 2 in Section 2). This also implied that different host with the same architecture and configuration can use the same calibration and consistently generate the same accurate CPU workload for different average execution times ranging between [1, 1000] msec.

Validating the emulation technique against multiple threads of execution. Distributed system developers understand that as components of the SLICE scenario are collocated, their the execution time (or service time) for handling events will increase. This is due to having separate threads of execution for processing each event and hardware/software contention on the host. Distributed system developers of the SLICE scenario expect to experience similar behavior when using the CPU workload generator for early integration testing. They therefore hypothesize that the emulation technique realized in CUTS’s CPU workload generator will produce execution times greater than the expected execution time depending on the number of threads executing on the host.

Figure 7 presents the results for emulation 3 different threads of execution to single processor. As highlighted in Figure 7, each thread has an expected execution time: 30 msec (top), 70 msec (middle), and 120 msec (bottom). Figure 7 illustrates, however, that the measured execution time is greater than the specified execution time. This is because the CUTS’s CPU workload generator is emulating CPU time instead of wall time. Moreover, if software performance engineering [18] techniques are taken into account, then the measured execution time for the emulation is bounded by Equation 1:

$$S_i < D_i < n \times S_i \quad (1)$$

where D_i is the measured service time (or service demand), n is the number of threads executing on the host, and S_i is the expected execution time (or service time) of the thread.

Because of the results presented in Figure 7, distributed system developer’s hypothesis about the behavior of CUTS’s CPU workload generator for multiple threads of execution was correct. More importantly, they have an emulation technique that will accurately emulate CPU workload, and produce realistic results when collocating components of the SLICE scenario.

4.2 Evaluating Performance of the SLICE Scenario

In Section 4.1, distributed system developers validated the emulation technique and capabilities of CUTS’s CPU workload generator. Moreover, results showed that CUTS’s CPU workload generator produces realistic behavior in environments with multiple

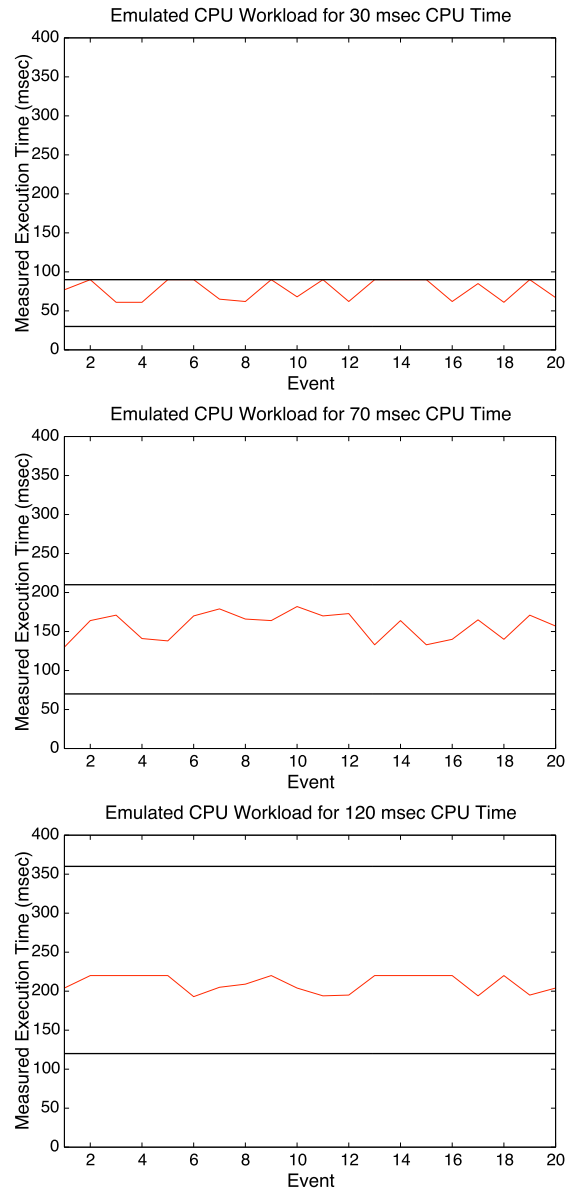


Fig. 7. Measured CPU workload for three threads of execution on same processor.

threads of execution, such as collocating components of the SLICE scenario. Distributed system developers therefore plan to use the CUTS’s CPU workload generator to evaluate the critical path of the SLICE scenario on different high-performance computing architectures.

In particular, distributed system developers plan to evaluate the improvement in performance of the critical path for the SLICE scenario when all components are collocated on the same host using different high-performance computing architectures. This will enable them to understand the side-effects of such architectures and can provide valuable insight in the future, such as determining how many hosts will be needed to ensure optimal performance of the SLICE scenario. They therefore used a single host in ISISlab to conduct several experiments.

Table 1. Emulation results of SLICE scenario on different architectures.

Architecture	Avg. Exec. Time (msec)
Single	98.09
Dual-core	87.61
Dual-core (hyper-threaded)	65.8

Table 1 presents results of a single experiment that measured average end-to-end response time of the SLICE scenario’s critical path. The experiment was executed on three different configurations/architectures of a single node in ISISlab. As highlighted in Table 1, the dual-core with hyper-threading had the best performance of the three, which distributed system developers of the SLICE scenario expected. More importantly, however, the results presented in Table 1 show that CUTS’s CPU workload generator enabled distributed system developers of the SLICE scenario to construct realistic experiments and observe the effects of different high-performance computing architecture configurations during early phases of the software lifecycle.

5 Related Work

MinneSPEC [15] and Biesbrouck et. al [1] present benchmark suites for generating CPU workload. It is designed to generate computation workload for new high-performance computing architectures being simulated. The CPU workload generation technique discussed in this paper differs from MinneSPEC and Biesbrouck’s work in that it abstracts away CPU characteristics that their benchmarks target in its computation workload. Moreover, the emulation technique discussed in this paper is designed to evaluate application-level performance and MinneSPEC and Biesbrouck’s work is designed to evaluate architecture-level performance. It is believed, however, that the computation workload in MinneSPEC and Biesbrouck’s work can be used as the abstract computation workload for the emulation technique presented in this paper.

Jeong et. al [14] present a technique for emulating CPU-bound workload in the context of designing database system benchmarks. Their CPU workload emulation technique uses a simple arithmetic computation that is continuously executed based on the number

of times specified in the benchmark. The emulation technique in this paper extends their approach by representing similar arithmetic computations as abstract computations and accurately emulating them from [0,1000] msec. This enables distributed system developers to construct more controlled experiments when conducting system integration test during early phases of the software lifecycle.

6 Concluding Remarks

Evaluating enterprise DRE system performance on different high-performance computing architectures during early phases in the software lifecycle enables distributed system developers to make critical choices about system design and the target architecture before it is too late to change. Moreover, it enables them to identify performance bottlenecks before they become too costly to locate and rectify. This paper therefore presented a technique for emulating computation intensive workload independent of the underlying high-performance computing architecture during early phases of the software lifecycle. Based on the experience gained from applying the emulation technique for computation intensive enterprise DRE system, the following is a list of lessons learned:

- **Abstraction via emulation techniques simplifies construction of early integration test scenarios.** Instead of wrestling with low-level architecture concerns, distributed system developers focus on the overall execution times of the enterprise DRE system. This enables developers to create more realistic early system integration test with little effort, and concentrate more on evaluation the system under development on its target architecture.
- **Lack of details makes it hard to compare computation workload across heterogeneous architectures.** Different architectures have different characteristics, such as processor speed and type. Moving a computation from one architecture to another will yield different relative performance. Future work therefore includes enhancing the emulation techniques so computation intensive workloads have valid relative execution times when migrated between heterogeneous architectures.
- **Abstraction reduces the complexity of accurately emulating computation intensive workload.** This is because it does not rely on low-level, tedious, error-prone, and non-portable techniques, such as constantly sampling the clock and querying hardware (or software) for thread execution time.

CUTS and the CPU workload generator discussed in this paper are available for download in open-source format at the following location: www.dre.vanderbilt.edu/CUTS.

7 Acknowledgements

Acknowledgements are given to Gautam Thaker from Lockheed Martin Advanced Technology Labs (ATL) in Cherry Hill, NJ. His knowledge of emulating CPU workload for enterprise DRE systems and critique on the technique and algorithm presented in this paper is greatly appreciated.

References

1. M. V. Biesbrouck, L. Eeckhout, and B. Calder. Representative Multiprogram Workloads for Multithreaded Processor Simulation. In *IEEE 10th International Symposium on Workload Characterization*, pages 193–203, September 2007.
2. S. Bohacek, J. Hespanha, J. Lee, and K. Obraczka. A hybrid systems modeling framework for fast and accurate simulation of data communication networks. In *Proceedings of ACM SIGMETRICS '03*, June 2003.
3. J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *International Journal of Computer Simulation, Special Issue on Simulation Software Development Component Development Strategies*, 4, Apr. 1994.
4. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.
5. G. de A Lima and A. Burns. An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems. *Computers, IEEE Transactions on*, 52(10):1332–1346, Oct. 2003.
6. A. Haghighat and M. Nikravan. A Hybrid Genetic Algorithm for Process Scheduling in Distributed Operating Systems Considering Load Balancing. In *In Proceedings of Parallel and Distributed Computing and Networks*, February 2005.
7. M. Hauswirth, A. Diwan, P. F. Sweeney, and M. C. Mozer. Automating Vertical Profiling. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 05)*, pages 281–296, New York, NY, USA, 2005. ACM Press.
8. J. H. Hill and A. Gokhale. Model-driven Engineering for Early QoS Validation of Component-based Software Systems. *Journal of Software (JSW)*, 2(3):9–18, Sept. 2007.
9. J. H. Hill and A. Gokhale. Model-driven Specification of Component-based Distributed Real-time and Embedded Systems for Verification of Systemic QoS Properties. In *Proceeding of the Workshop on Parallel, Distributed, and Real-Time Systems (WPDRTS '08)*, Miami, FL, April 2008.
10. J. H. Hill and A. Gokhale. Towards Improving End-to-End Performance of Distributed Real-time and Embedded Systems using Baseline Profiles. *Software Engineering Research, Management and Applications (SERA 08), Special Issue of Springer Journal of Studies in Computational Intelligence*, 150(14):43–57, Aug. 2008.
11. J. H. Hill, J. Slaby, S. Baker, and D. C. Schmidt. Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded System QoS. In *Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications*, Sydney, Australia, August 2006.
12. M. D. Hill. Opportunities Beyond Single-core Microprocessors. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 97–97, New York, NY, USA, 2008. ACM.
13. S. E. Institute. Ultra-Large-Scale Systems: Software Challenge of the Future. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, Jun 2006.
14. H. J. Jeong and S. H. Lee. A Workload Generator for Database System Benchmarks. In *Proceedings of the 7th International Conference on Information Integration and Web-based Applications & Services*, pages 813–822, September 2005.
15. A. KleinOowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *IEEE Computer Architecture Letters*, 1(1):7, 2002.

16. Á. Lédeczi, Á. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, 2001.
17. M. W. Masters and L. R. Welch. Challenges For Building Complex Real-time Computing Systems. *Scientific International Journal for Parallel and Distributed Computing*, 4(2), 2001.
18. D. A. Menasce, L. W. Dowdy, and V. A. F. Almeida. *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.
19. Rittel, H. and Webber, M. Dilemmas in a General Theory of Planning. *Policy Sciences*, pages 155–169, 1973.
20. C. Smith and L. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley Professional, Boston, MA, USA, September 2001.
21. B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.