ETI Technical Report 01

Landing Containment Domains on SWARM: Toward a Robust Resiliency Solution on A Dynamic Adaptive Runtime Machine

Sam Kaplan
Sergio Pino
Guang R. Gao

Abstract

Software and hardware errors are expected to be a much larger issue on exascale systems than current hardware. For this reason, resilience must be a major component of the design of an exascale system. By using containment domains, we propose a resilience scheme that works with the type of codelet-based runtimes expected to be utilized on exascale systems. We implemented a prototype of our containment domain framework in SWARM, and adapted a Cholesky decomposition program written in SWARM to use this framework. We will demonstrate the feasibility of this approach by showing the low overhead and high adaptability of our framework.

Introduction

Exascale systems are expected to exhibit a much higher rate of faults than current systems, for a few reasons. Given identical hardware, failure rate will increase at least linearly with number of nodes in a system. In addition, exascale hardware will include more intricate pieces, including smaller transistors, which will be less reliable due to manufacturing tolerances and cosmic rays. Software will also have increased complexity, which again results in more errors. [1] The combination of the above factors indicates that resilience will be incredibly important for exascale systems, to a higher degree than it has for any preceding generation of hardware.

On current systems, most resilience methods take the form of checkpointing. Common types of checkpointing exhibit flaws that limit their scalability to exascale, due to the larger amount of state needing to be saved, and the lower mean time between failures. For this reason, it is desirable to have a resilience scheme that requires no coordination and can scale to any workload size. To this end, we leveraging ideas from containment domain research performed by Mattan Erez and his team at University of Texas at Austin [2]. Similar to codelet model used in SWARM, containment domains exhibit a distributed, fine-grained, hierarchical nature. For this reason, we expect the impact of containment domains to be well realized when mapping onto a codelet model. We hope to show the feasibility of this approach by implementing containment domains in SWARM, using a continuation-based API.

SWARM (SWift Adaptive Runtime Machine) is a codelet-based runtime created at ETI. We have previously adapted applications to use fine-grained, distributed, low-overhead SWARM codelets, and have demonstrated positive results in both performance and scalability. Because

of its efficiency, maturity, and programmability, as well as our own familiarity with it, SWARM was chosen as the underlying runtime for our resilience research.

By implementing a prototype containment domain framework in SWARM, we show the feasibility of utilizing containment domains in a codelet-based runtime. Specifically, we created a continuation-based API to allow containment domains to conform to the requirements of the codelet model: fine-grained, non-blocking, and largely self-contained. We adapted a Cholesky decomposition program written in SWARM to use this API, showing that necessary functionality is implemented and performs correctly. We also benchmarked this program to show that our implementation of containment domains has a very low overhead.

Background

Containment domains were first proposed by Mattan Erez at University of Texas at Austin. Described as a "scalable and efficient resilience scheme" [3], containment domains are a possible solution to the increased error rate expected to be seen on exascale systems. By allowing data preservation and recovery in an application-specific manner, containment domains allow for uncoordinated, low-overhead resilience.

At a high-level, a containment domain contains four components: data preservation, to save any necessary input data; a body function which performs algorithmic work; a detection function to identify hardware and software errors; and a recovery method, to restore preserved data and re-execute the body function. The detection function is a user-defined function that will be run after the body. It may check for hardware faults by reading error counters, or for software errors by examining output data (e.g. using a checksum function). Since containment domains can be nested, the recovery function may also escalate the error to its parent. Since no coordination is needed, any number of containment domains may be in existence, with multiple preserves and recoveries taking place simultaneously.

An initial prototype implementation of containment domains was developed by Cray [4]. In addition, a more fully-featured containment domain runtime is in currently in development by Mattan Erez and his team. However, none of these implementations support a continuation-based model. If exascale hardware is going to use a codelet-based runtime, it is necessary to adapt these ideas to support such a model. For this reason, it is important that we demonstrate use of a codelet-based runtime, in this case SWARM.

Containment Domains in SWARM

We developed our containment domain API as an additional feature of the SWARM runtime. This allows us to leverage existing runtime features and internal structures in order to support the hierarchical nature of containment domains. The main features needed include data preservation, user-defined fault detection functions, and re-execution of failed body functions. This feature set was realized by implementing the following functions:

`swarm_ContainmentDomain_create(parent)`: Create a new containment domain as a child of `parent`.

`swarm_ContainmentDomain_begin(THIS, body, body_ctxt, check, check_ctxt, done, done_ctxt)`: Begin execution of the current containment domain (`THIS`) by scheduling the `body` codelet with `body_ctxt` as its context parameter. The `begin` codelet is scheduled with its `NEXT` parameter set to a callback within the SWARM runtime, which will schedule the `check` codelet with `check_ctxt` as its context. In the `check` codelet, `NEXT` is again set to a SWARM-internal codelet, which takes a boolean success value as the `INPUT` parameter. If this value is `TRUE`, the `done` codelet is scheduled, with `done_ctxt` as its context. Codelet chaining from `begin` and `check` is supported, provided the user preserves the original `NEXT` parameter and schedules it from every end point in the chain.

`swarm_ContainmentDomain_preserve(THIS, data, length, id)`: In the specified containment domain (`THIS`), do a memory copy of `length` bytes from `data` into a temporary location inside the CD. We support multiple preservations per CD (e.g. to allow preservation of tiles within a larger array, such that the individual tiles are non-contiguous in memory), by adding a user-selected `id` field. For each containment domain in SWARM, a boolean value is set based on its execution status. On first execution, data is preserved normally. On subsequent executions, data is copied in reverse (i.e. from the internal preservation into the `data` pointer).

`swarm_ContainmentDomain_finish(THIS)`: Close current containment domain (`THIS`), discard all of its preserved data, and make its parent active.

Example

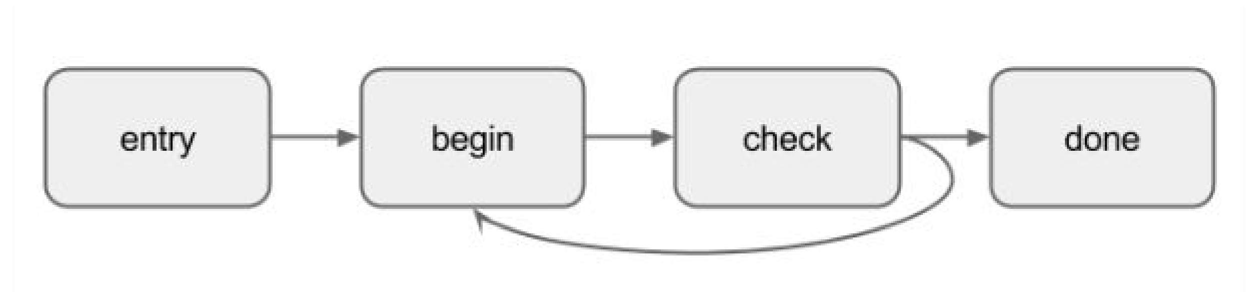

Figure 1: Simple CD application

Figure 1 shows a graph of a very simple program using containment domains. This example shows a small program that multiplies two integers, and uses a single containment domain. The `entry` codelet initializes the containment domain, and enters it. The `begin` codelet multiplies its two inputs, and stores the result in `C`. The `check` codelet performs the same multiplication, and compares with the original result in `C`. If the results are not the same, an error has occurred and must be corrected. The `begin` codelet is re-executed, and the inputs are recovered from their saved locations. This continues until the `begin` and `check` codelets achieve the same result, in which case the `done` codelet is called, and the runtime is terminated.

Results

The following observations were made as a result of our research.

Observation 1: Feasibility

It is feasible to adapt a codelet-based application to use containment domains. This can be seen from the implementation of our framework in SWARM, and our working Cholesky application using said framework.

Observation 2: Efficiency

Our implementation has a very low overhead. As shown in Figures 2 and 3, the relative overhead of a containment domain is dependent on the amount of work performed inside it. If the workload is large enough, the overhead from adding containment domains is negligible.

Observation 3: Resilience

As shown in Figure 4, it is possible to simulate errors, and have the containment domain framework re-execute codelets as necessary to achieve the correct result.

To show that our prototype implementation has sufficient functionality and efficiency, we instrumented a Cholesky decomposition program in SWARM to use containment domains. This was based on a Cholesky implementation included as an example in the SWARM distribution.

The Cholesky program has only three main codelets, one for each linear algebra routine run on a tile (POTRF, TRSM, and GEMM/SYRK), and each of these is called a number of times for each tile. For our purposes, each of these is considered a containment domain. In our model, we only considered faults similar to arithmetic errors; that is, incorrectly calculated results. For this reason, we did not need to preserve input data unless it would be overwritten by an operation (e.g. the input/output tile for a POTRF operation). In order to simulate arithmetic errors, rather than relying on error counters from actual faulty hardware, a random number generator was used. If a random number was below a configurable threshold, a fault was deemed to have occurred.

For all of the following experiments, the program was run on a dual-processor Intel Xeon system, using 12 threads. Although the machine has 24 hyperthreads available, benchmarks showed that limiting the runtime to 12 threads gave the best performance for this program. The workload sizes were confirmed to not exhaust the physical memory of the machine. Since the execution time of the program naturally varies between runs (due to factors like scheduling differences), all of the included times are the average of 5 runs.

In Figure 2, we show the execution time for our Cholesky program with different tile sizes, with a constant matrix size. The overhead from our implementation of containment domains can be seen below. Data preservation accounts for the bulk of this overhead. Besides preservation, overhead comes from the additional codelets needed (check and done functions), and random number generation needed to determine success. These sources of overhead are separated from preservation overhead in the graph.

Less total data is preserved with fewer tiles (hence fewer iterations), since each iteration must preserve data from the rest of the matrix. For this reason, the time spent preserving data decreases sharply with a larger tile size. However, increasing the tile size too much results in less parallelism overall.

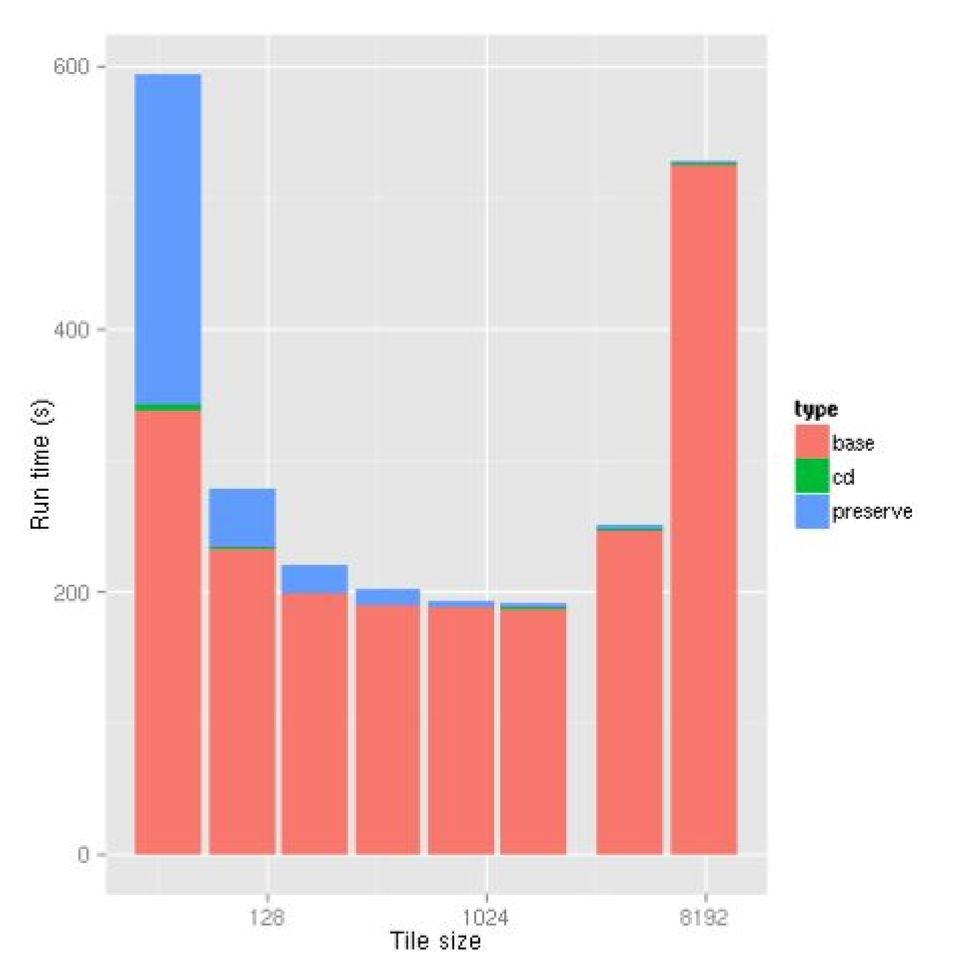

Figure 2: Cholesky execution time

Figure 3 shows the percentage overhead from containment domains (excluding preservation) for each tile size. Since the expected additional work is constant for each containment domain (scheduling and running the check and done codelets), regardless of tile size, the percentage overhead is larger for smaller tile sizes, for which less algorithmic work is performed in a containment domain. Except for the smallest tile sizes we tested, the variation is largely due to random scheduling perturbations (including one case where the program ran slightly faster with containment domains!). This shows that assuming enough work is done per containment domain, the overhead from our framework is negligible

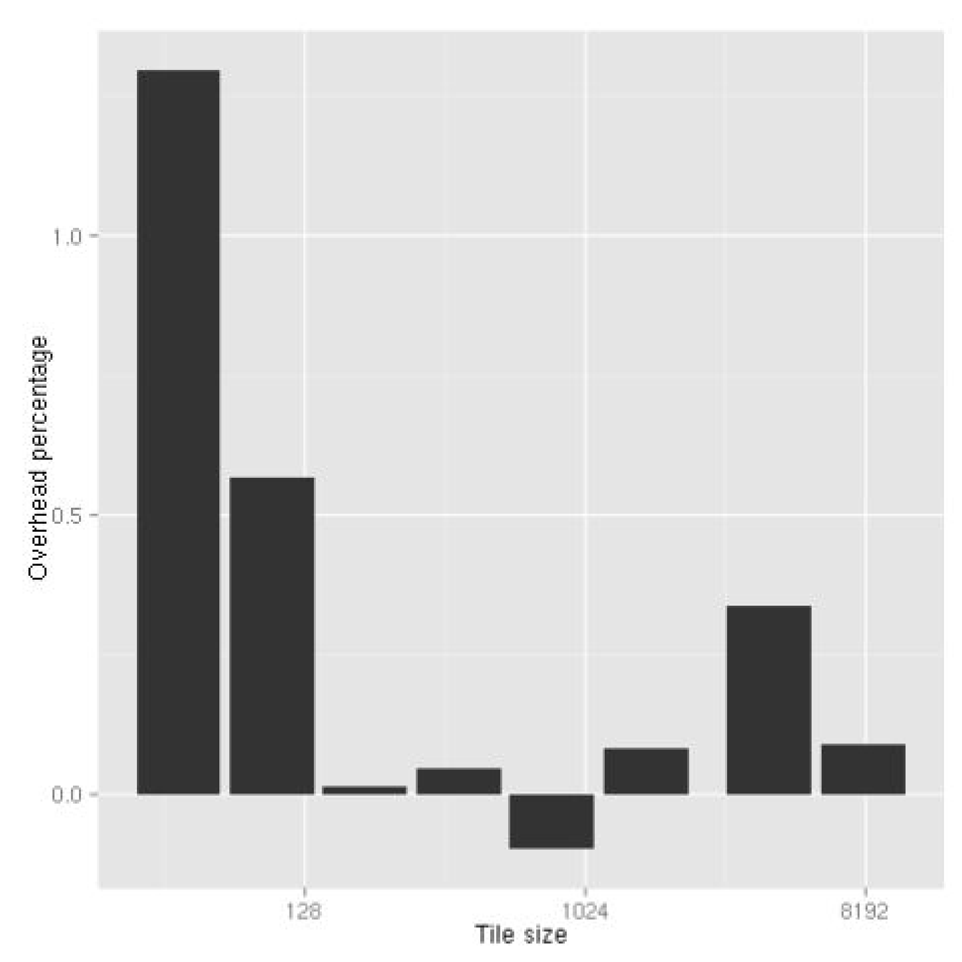

Figure 3: Percentage overhead vs. tilesize

We also examined the running time of our Cholesky program with different simulated error rates. Each of these experiments was performed with a 40000x40000 matrix (same as above), with tile size 200x200. As shown in Figure 4, the running time conforms reasonably well to the idealized case (i.e. an error rate of 0.5 results in a 2x execution time). The resulting time is slightly lower than expected because a small portion of code (to check readiness of codelet dependencies) is outside of containment domains and therefore not re-executed with the main body functions.

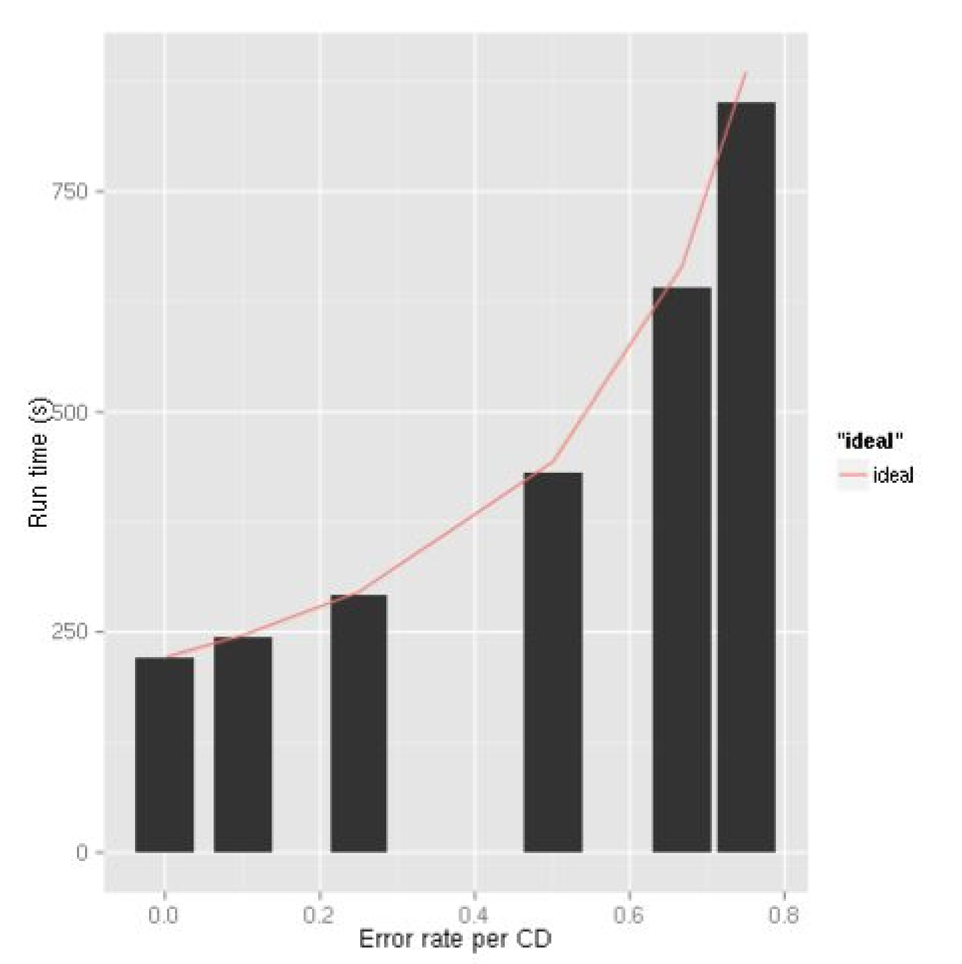

Figure 4: Execution time with simulated errors

Conclusions and Future work

In conclusion, we have demonstrated that containment domains can be adapted to the codelet model. Our Cholesky application shows that containment domains can be used in a decentralized, continuation-based manner, to provide a fine-grained, low-overhead framework for resilience. Although the best method for adapting individual applications is still an open problem, we have shown that this is an approach worth pursuing.

Our prototype requires the user to obey a few limitations, due to its limited feature set. Firstly, we only support single-node SWARM applications. We also only allow references to the inner-most CD on preservation. To avoid data duplication, a fully fleshed out implementation would allow preservation in an outer CD, which could be referred to by multiple child CDs. An obvious source of improvement would be to add support for these missing features.

Due to the Cholesky program's very decentralized call graph, it was not feasible to add nested containment domains in any useful manner. If we had another example, it would likely provide additional insight. We attempted to implement an SCF program, but due to the increased complexity of the application, we ran into issues completing this in time.

Sources

[1] Cappello, F., Geist, A., Gropp, W., Kale, S., Kramer, B., & Snir, M. (2014). Toward Exascale Resilience: 2014 update. *Supercomputing Frontiers And Innovations, 1*(1), 5-28. doi:http://dx.doi.org/10.14529/jsfi140101
[2] http://lph.ece.utexas.edu/public/CDs/ContainmentDomains
[3] Jinsuk Chung, Ikhwan Lee, Michael Sullivan, Jee Ho Ryoo, Dong Wan Kim, Doe Hyun Yoon, Larry Kaplan, and Mattan Erez. **Containment Domains: A Scalable, Efficient, and Flexible Resilience Scheme for Exascale Systems.** *In the Proceedings of SC'12*. November, 2012.
[4] http://craycontainment.sourceforge.net/index.html