An Automatic Actors to Threads Mapping Technique for JVM-based Actor Frameworks

Ganesha Upadhyaya
Iowa State University, USA
ganeshau@iastate.edu

Hridesh Rajan
Iowa State University, USA
hridesh@iastate.edu

Abstract

Actor frameworks running on Java Virtual Machine (JVM) platform face two main challenges in utilizing multi-core architectures, i) efficiently mapping actors to JVM threads, and ii) scheduling JVM threads on multi-core. JVM-based actor frameworks allow fine tuning of actors to threads mapping, however scheduling of threads on multi-core is left to the OS scheduler. Hence, efficiently mapping actors to threads is critical for achieving good performance and scalability. In the existing JVM-based actor frameworks, programmers select default actors to threads mappings and iteratively fine tune the mappings until the desired performance is achieved. This process is tedious and time consuming when building large scale distributed applications. We propose a technique that automatically maps actors to JVM threads. Our technique is based on a set of heuristics with the goal of balancing actors computations across JVM threads and reducing communication overheads. We explain our technique in the context of the Panini programming language, which provides capsules as an actor-like abstraction for concurrency, but also explore its applicability to other approaches.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming ; D.2.7 [*Distribution, Maintenance, and Enhancement*]: Portability ; D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Processors*]: Code generation, Optimization

General Terms Experimentation, Measurement, Performance, Scalability

Keywords Actors, Panini programming language, Capsule-oriented Programming, Implicit Concurrency, Multi-core, Java, JVM, Thread Mapping, CPU Utilization

AGERE! 2014, October 20, 2014, Portland, OR, USA..

http://dx.doi.org/10.1145/2687357.2687367

1. Introduction

Multicore and many-core systems have become the industry standards. The number of cores on these systems is rapidly increasing. Sequential programs are ill-suited to run on multicore systems as they do not utilize the available cores. Many concurrent and parallel programming models have emerged to utilize cores effectively. One such model is the Actor model [8] which has self contained concurrently runnable actors that communicate via message passing. Actor model exposes parallelism by design. The parallelism in actor systems stems from being able to execute multiple actors in parallel. The popularity of actor model has influenced programmers to build large distributed applications that perform data, task and pipeline parallelism at fine and coarse granularity.

However, for utilizing the underlying multicore, actors need to be mapped to cores carefully. Mapping is a two step process: 1) actors $\rightsquigarrow$ threads mapping and 2) threads $\rightsquigarrow$ cores mapping (or scheduling). Often, the actor language runtime handles both steps by creating required threads and scheduling them on different cores using an efficient actor to core mapping technique. However, in JVM-based actor languages, actors to threads mapping is performed by programmers and JVM leaves scheduling of threads on multicore to the OS scheduler. Programmers select default actors to threads mappings and iteratively fine tune the mappings until the desired performance is achieved. This process is easy for simple or embarrassingly parallel applications, however for applications that have sub-linear performance, improving the application performance is tedious and time consuming [21].

It can be argued that an optimal solution for actors to threads mapping using brute-force technique cannot be found in polynomial time (similar to the task mapping problem [21]). For this purpose, we propose a novel heuristic-based approach that automatically maps actors to threads at compile time. Our heuristics use actor characteristics such as lifetime of the actor, amount of computation, nature of computation, and nature of interaction with other actors in the system, etc. The central goal of the heuristics is to map actors to threads in a way that balances actor computational

workloads and reduces communication overheads. The outcome of applying the proposed heuristics in a systematic manner results in one of the four different execution policies (Thread, Task, Sequential, Monitor) for actors. The execution policy defines how the messages are processed. In thread execution policy messages are processed by a dedicated JVM thread. In task execution policy messages are processed by a shared thread and in sequential/monitor case the actor that sends message itself processes the message at the receiver actor. By means of assigning different execution policies to actors at compile-time we achieve actors to threads mappings.

We propose that programmers use our heuristic-based mapping technique for initially mapping actors to threads in place of default mapping technique. We believe that by using our heuristic-based mapping technique programmers time and efforts to arrive at initial optimal mapping are saved.

We have realized our technique in the Panini programming language [1, 20]. Panini programming language provides an abstraction for implicit concurrency called *capsules*, which are a flavor of actors for sequentially trained programmers[1]. Even though, our technique is well-suited for Panini's capsules where programmers can concentrate their efforts in application development rather than understanding and improving concurrent execution, it is applicable to any JVM-based actor frameworks. Our evaluation over a wide range of actor programs shows that our heuristic-based mapping achieves on average 50% improvement in program runtime over default-thread and default-task mappings.

2. Related Work

Actor frameworks such as Akka [15], Kilim [25], Scala Actors [16], Jetlang [5], ActorFoundry [10], SALSA [12] and Actors Guild [2] allow programmers to map their actors to JVM threads and fine tune their application using schedulers and dispatchers. In Akka, dispatchers are responsible for actor scheduling. Akka provides four kinds of dispatchers: default, pinned, balancing, and calling thread. The default dispatcher is an event-based single queue implementation backed by a thread-pool of configurable size. In pinned dispatcher each actor has it own OS thread. The balancing dispatcher employs event-based approach and uses single mailbox and performs load-balancing. Finally, the calling thread dispatcher has no thread associated with it and the thread of the message sender executes the message. Kilim is an actor framework for Java. It provides lightweight event-based actors. Kilim scheduler is a bundle composed of a thread-pool, a scheduling policy and collection of runnable actors. By default, actors in the collection are scheduled in a round-robin fashion. Scala Actors allow creation of thread-based and event-based actors. Thread-based actors are assigned dedicated JVM threads, whereas event-based actors share JVM threads associated with the task-pool that processes event-based actor messages. For the purpose of fair scheduling, task-pool uses round-robin scheduling. ActorFoundry internally uses kilim and Actors Guild follows similar strategy as Scala Actors. SALSA allows creation of heavy-weight and light-weight actors using Stage. Stage is a bundle composed of a single message queue and a JVM thread. Heavy-weight actors are assigned individual stages and light-weight actors share stages. However, mapping actors to stages is left to the application developers.

In all these actor languages and frameworks, application developers define actors to threads mappings and fine tune the mapping until the desired performance is achieved. Whereas, our technique performs automatic mapping of actors to threads.

Several works on performance improvement of non-JVM actor frameworks exists. Francesquini *et al.*[13] proposed a technique implemented in Erlang [27] runtime that places Erlang actors on multi-core efficiently. Their technique showed that by placing frequently communicating actors (hub-and-affinity) together, over two times improvement in the application performance can be achieved. However, programmers need to identify hub and its affinity actors and annotate the program for runtime to perform the desired mapping. Our technique uses many more actor characteristics along with hub-and-affinity and performs essential program analyses to automatically determine the mappings.

Mapping application on to multi-core is well studied problem. The application is represented as task graph and the mapping problem is defined as how to map different tasks to CPU cores to minimize application runtime. A recent survey by Singh *et al.* [22] lists different static, dynamic and hybrid techniques that map task graph to multi-core with performance, energy consumption and temperature as different goals of determining optimal mapping. Researchers have explored the problem of mapping application tasks that communicate via both message passing and shared memory on homogeneous and heterogeneous cores [24]. These techniques are not directly applicable to JVM-based actor frameworks, because threads to cores mapping is left to OS scheduler and only actors to threads mapping can be optimized. However, actors to threads mapping technique can utilize solutions proposed for general task graph mapping problem. In our heuristic-based mapping technique, we utilize actor characteristics and interaction behaviors similar to task characteristics and task graph in general task graph mapping problem.

Note that the mapping problem in actor programs is different from the mapping problem in general multi-threaded programs. In multi-threaded programs, the mapping problem is defined as scheduling and load-balancing of threads on multi-cores. This also involves binding of threads to physical cores. However in actor programs, the mapping problem is two-fold: mapping actors to threads and scheduling

[1] Main promise of capsules is that they can enable modular reasoning about concurrent programs, but that direction is explored elsewhere [11].

of threads on multi-cores. Tousimojarad *et al.* [26]'s work proposes efficient strategies for mapping threads to cores for OpenMP multi-threaded programs. When compared to this work, our technique maps actors to threads and not threads to cores. Threads to cores mapping is handled by OS scheduler in JVM-based actor frameworks.

3. Actors to JVM Threads Mapping Problem

Traditionally actors tend to have a dedicated thread that processes messages in their queue. As the popularity of actor programming paradigm grows, actor languages are used to develop large scale distributed applications. These applications spawn a large number of actors and hence dedicating a thread for every actor is not scalable. Actor languages have realized this limitation and allowed programmer to share a thread across multiple actors. However, it is the responsibility of the programmer to correctly map actors to threads.

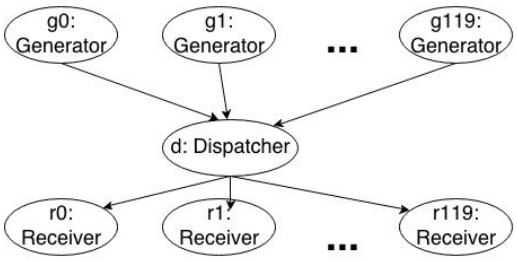

Figure 1. BenchErl serialmsg actor communication graph.

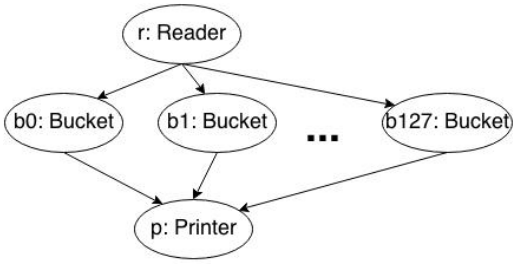

Figure 2. Panini histogram actor communication graph.

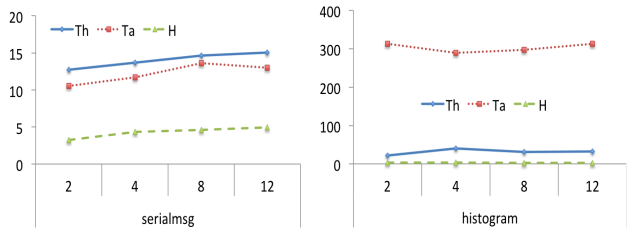

Figure 3. Performance of default-thread and default-task mappings for BenchErl serialmsg and Panini histogram benchmark programs (x-axis: 2, 4, 8, and 12-core settings and y-axis: runtime in seconds).

To illustrate the problem consider mapping actors to threads for two actor programs shown in Figure 1 and Figure 2. Our first program shown in Figure 1 is the serialmsg program from BenchErl benchmark suite [9]. This benchmark is about message proxying through a dispatcher and spawns a certain number of receivers, one dispatcher, and a certain number of generators. The dispatcher forwards the messages that it receives from generators to the appropriate receiver. Each generator sends a number of messages to a specific receiver. The second program shown in Figure 2 is the histogram program from Panini [1, 20]. The goal of this problem is to count the number of times each ASCII character occurs on a page of text.

For deciding initial actors to threads mapping programmers can use default-thread or default-task mappings. In default-thread mapping, every actor instance is assigned a dedicated thread and in default-task mapping, actors are mapped to a taskpool containing fixed number of threads (usually equals to number of CPU cores). Figure 3 shows performance of two benchmark programs on 2, 4, 8, and 12-core machines. The two main concerns are visible in the performance results. These are i) performance is not consistent across programs indicating that a single default mapping strategy (thread or task) does not work across programs, and ii) performance degrades upon adding more cores.

Now, the programmer can use default-thread or default-task mappings, profile the default mappings and iteratively fine tune the actors to threads mappings. However, when multiple actors are mapped to a JVM thread, profiling the thread and understanding the bottlenecks is difficult. We believe that some aspects of the actor model can be directly used to decide the initial actors to threads mappings. For instance, in the illustrative program shown in Figure 1, generators and receivers can be assigned to single thread to avoid message passing overheads. For addressing the scalability problem, choosing the right number of JVM threads plays an important role. The solution of limiting number of threads works for applications that have smaller number of actors. The challenge remains for applications with large number of actors. Hence the fundamental problem is finding an initial optimal mapping that has consistent performance and scales well when additional cores are allocated.

3.1 Problem Formulation

We formulate the problem of mapping actors to JVM threads as selecting different execution policies for actors in the system. Execution policy defines how actor messages are processed. We define four different execution policies as follows,

- *THREAD*, in this execution policy, a dedicated thread is assigned for processing the messages from the actor's message queue and executing the corresponding behavior (works similar to akka's pinned dispatcher).
- *TASK*, in this execution policy, the actor messages are processed by the shared thread of the taskpool. The taskpool may contain one or more actors that abide to *TASK* execution policy. The order in which the messages

from different actors message queue has to be processed could vary. One simple policy is to process one message from each actor to avoid starvation of other actors.

- *SEQ/MONITOR*, in case of *SEQ* and *MONITOR*, the policy is that the calling actor that sends the message needs to execute the defined behavior at the callee actor that received the message (works similar to akka's calling thread dispatcher).

Actor communication graph (ACG). Given an actor program, actor communication graph defines various actors in the program and the interaction between them. ACG is a directed graph G(V,E) where,

- $V = A_0, A_1, ..., A_n$ is a set of nodes, each representing an actor.
- E is a set of edges (A_i, A_j) for all i,j such that there is communication from A_i to A_j.

Mapping function. Given actor program and ACG we define actors to threads mapping as assigning execution policies from the set of possible execution policies, $M(P \times ACG) \mapsto EP$ where,

- P is actor program,
- *ACG* is actor communication graph and,
- *EP* = {*THREAD*, *TASK*, *SEQ*, *MONITOR*}.

4. Approach

In this section we describe our heuristic-based mapping technique that assigns execution policies to actors. We first describe different aspects of actor programs that are relevant for our heuristics and formulate them as a *Actor Characteristics Vector (cVector)*. We then describe a set of heuristics that given *cVector* predicts the execution policy. Finally, we describe our mapping function followed by a set of examples illustrating the application of our mapping technique.

4.1 Analyzing Applications

Some aspects of the actor applications can be directly used to decide the execution policies for actors. The knowledge of the actor behavior and their communication graph that defines the relationship between the actors can be easily extracted. This subsection presents several of these aspects.

- blocking, an actor has blocking behavior if it has externally blocking behaviors using I/O, socket or database blocking primitives.
- stateful/stateless, actors may have state variables that are modified in multiple actor behaviors.
- inherent parallelism, actors may use blocking send primitives and receive results or use asynchronous send primitives. Actors may or may not require the results immediately.
- computationally intensive, actors may have different computational requirements.
- communication behavior (or message rate), actor system may contain leaf actors that do not send messages to other actors, or routing actors that sends exactly one message for every message it receives, or broadcast actors that sends multiple messages for every message it receives.
- hub-affinity, some actors communicates more with a set of actors than other actors in the system. These actors form hub-affinity group.
- data rate, some actors in the system may send/receive high volume of data.
- contention, some actors may have a tendency to receive messages from multiple competing actors.

4.2 cVector: Actor Characteristics Vector

Based on the different aspect about actor behaviors and their communication graph every actor is assigned a Characteristics vector. The Characteristics vector has five fields,

$$\langle\langle\ BLK,\ STATE,\ PAR,\ COMM,\ CPU\ \rangle\rangle$$

1. *BLK* = {*true*, *false*} represents blocking behavior,
2. *STATE* = {*true*, *false*} represents stateful/stateless behavior,
3. *PAR* = {*low*, *med*, *high*} represents inherent parallelism and values are assigned as follows,
   - *low*, if actor sends a synchronous message and waits for the result or consumes the result right away,
   - *high*, if actors sends an asynchronous message and does not require result,
   - *med*, otherwise
4. *COMM* = {*low*, *med*, *high*} represents communication behavior and it is determined as follows,
   - *low*, does not send messages to other actors,
   - *high*, sends more than one message to the connected actors,
   - *med*, sends exactly one message to the connected actor,
5. *CPU* = {*low*, *high*}, represents computational workload of the actor and it is determined as follows,
   - *high*, when recursive, loops with unknown bounds, makes high cost library calls,
   - *low*, otherwise

Actor characteristics vector is computed using actor program (*P*) and actor communication graph (*ACG*) by performing a number of program analyses. For determining blocking, we look for usage of external blocking primitives. We perform intra-procedural analysis of actor behavior definitions to determine stateful/stateless behavior, inherent paral-

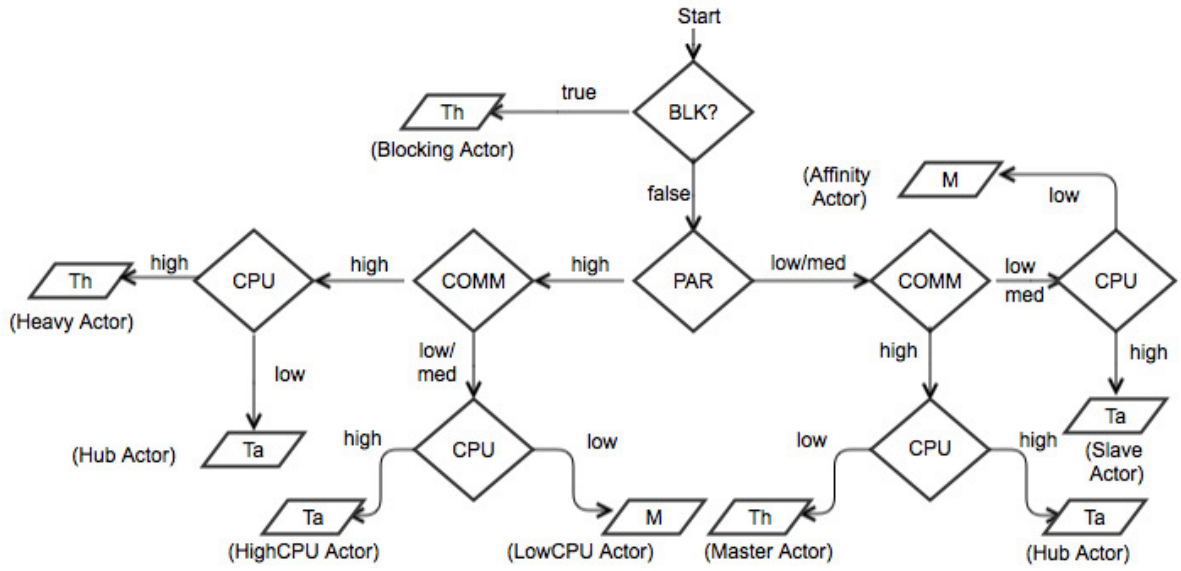

Figure 4. Flow diagram of our mapping function that assigns actors one of the four execution policies.

lelism (*PAR*), communication behavior (*COMM*) and computational requirement (*CPU*). We mark an actor stateful if more than one actor behavior accesses any of its state variable. Both *PAR* and *COMM* requires analysis of statements that involves sending messages. Note that in determining various fields of *cVector* the program analyses requires actor definition (actor code) and the communication behaviors of outgoing messages rather than incoming messages. This requirement helps our analyses to be less-strict about availability of complete *ACG* at compile time.

4.3 Mapping Heuristics

Earlier we presented four different execution policies that can be assigned to actors. This section examines a number of heuristics for predicting the execution policy. The heuristics make use of the actor characteristics vector *cVector*.

Blocking Actors Heuristics. This heuristic states that any actor that has external blocking behavior, as represented by BLK field in cVector, should be assigned thread (Th) execution policy. Any other execution policy for blocking actors would lead to blocking of the executing thread and may lead to actor starvation and deadlocks. The cVector for such actors is <true,_,_,_,_>. If *BLK* is *true*, other fields of cVector are ignored in making the execution policy decision.

Heavy Actors Heuristics. This heuristic states that any actor that is non-blocking with high inherent parallelism, high communication and high computation should be assigned thread execution policy. The cVector of such an actor is <false,_,high,high,high>. The rationale behind this decision is that the assigned thread can perform its CPU intensive computations in parallel with other threads without voluntarily interruption.

HighCPU Actors Heuristics. Actors that have high inherent parallelism with high CPU needs but low communication frequency are assigned task (Ta) execution policy. These actors have cVector <false,_,high,low,high>. By assigning task execution policy, these actors can utilize any load-balancing strategies applied to the task-pool.

LowCPU Actors Heuristics. LowCPU actors that have characteristics vector cVector <false,_,high,low,low> should be assigned monitor (M) execution policy. These actors do not need special attention and hence are processed by the calling actor (actor that sends messages).

Hub Actors Heuristics. This heuristic states that hub actors should be assigned task (Ta) execution policy. Hub actors are represented using cVector either <false,_,high,high,low> or <false,_,low/med,high,high>. The rationale behind this decision is that affinity actors (actors that hub actor communicates often) can be executed by the shared thread that is executing the hub actor task in the task-pool.

Affinity Actors Heuristics. This heuristic states that affinity actors should be assigned monitor (M) execution policy. Affinity actors have following cVector <false,_,low/med ,low/med,low>. By assigning monitor execution policy, the hub actor (of these affinity actors) is forced to execute the affinity actors.

Master Actors Heuristics. This heuristic states that master actors should be assigned thread (Th) execution policy. The cVector of master actors is <false,_,low/med,high,low>. Master actors have the property that they delegate the work to slave actors and often wait for the result. Hence, by assigning a dedicated thread it does not block the execution of slave actors.

Slave Actors Heuristics. This heuristic states that slave actors should be assigned task (Ta) execution policy. The cVector of slave actors is <false,_,low/med,low/med,high>. Similar to HighCPU actors these actors can utilize any load-balancing strategies applied to the task-pool.

4.4 Mapping function

Figure 4 describes the flow of our mapping function. For every actor in the system, mapping function assigns one of the four execution policies. By following the flow it is easy to see that the strategy is complete. Because, every actor with a cVector is assigned an execution policy. It can also be seen that actors are never assigned multiple execution policies.

4.5 Examples

We have implemented our technique in Panini Capsules (an actor flavor), hence we first briefly describe Panini Capsules. We then present several example panini programs and demonstrate the application of our mapping technique to decide execution policies. The panini source code of these programs is available in [6].

4.5.1 Panini Capsules

Capsules are an actor-like abstraction implemented in the programming language Panini [1, 11, 20]. Figure 5 presents an example HelloWorld program in this language. In this program there are three actors (capsules) HelloWorld, Greeter and Console and they are connected as HelloWorld → Greeter → Console.

```
1  signature Stream { //A signature declaration
2    void write(String s);
3  }

5  capsule Console () implements Stream { //Capsule declaration
6    void write(String s) {  // Capsule procedure
7      System.out.println(s);
8    }
9  }

11 capsule Greeter (Stream s) { //Requires an instance of Stream to work
12   String message = "Hello World!"; //  State declaration
13   void greet(){                    // Capsule procedure
14     s.write("Panini: " + message); // Inter−capsule procedure call
15     long time = System.currentTimeMillis();
16     s.write("Time is now: " + time);
17   }
18 }

20 capsule HelloWorld() {
21   design {          // Design declaration
22     Console c; // Capsule instance declaration
23     Greeter g;  // Another capsule instance declaration
24     g(c);         // Wiring, connecting capsule instance g to c
25   }
26   void run() {  // An autonomous procedure
27     g.greet();  // Inter−capsule procedure call
28   }
29 }
```

Figure 5. HelloWorld Program in Panini

In Panini's programming model, capsules are independently acting entities. Capsules provide interface to communicate to other capsules via capsule procedures. When a capsule wants to communicate with other capsule it does so using inter-capsule procedure calls. In the HelloWorld program described above, *g.greet()* in line 27 is an inter-capsule procedure call between HelloWorld capsule and Greeter capsule. If a capsule requires return result of inter-capsular call then the caller receives a future as a proxy for the actual return value (void return values are allowed). If the value is not used immediately, the caller can continue execution. Inter-capsule procedure calls are processed using message passing mechanism similar to actors. There are two kinds of capsules: those that define autonomous behavior by declaring a run procedure, and those that respond to request from other capsules.

4.5.2 BenchErl Bang

Bang benchmark simulates many-to-one message passing. It spa-wns one receiver and multiple senders that flood the receiver with messages. There are 440 instances of Sender and one instance of Receiver. Each Sender sends 440 messages to Receiver. The actor communication graph (ACG) is shown in Figure 6.

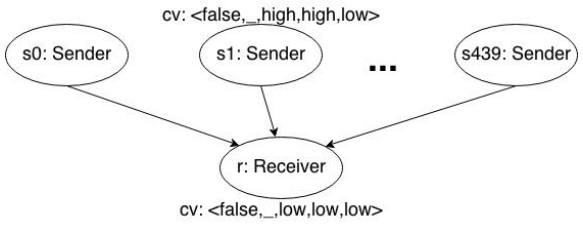

Figure 6. ACG of BenchErl Bang program.

Sender capsule has the characteristics vector (cv): <false,_, high,high,low>, because it is non-blocking, it has high inherent parallelism as inter-capsular call does not expect the results, it highly communicates with Receiver capsule and it does not have CPU intensive computation in it. Hence, by applying our mapping function shown in Figure 4, we determine that Sender capsule instances should be assigned *Task* execution policy. The cV for Receiver is <false,_,low,low,low> and by following the mapping strategy, it is assigned *Monitor* execution policy. A general intuition is to assign *Task* execution policy to Receiver, because it processes large number of messages from Sender capsules. However, if assigned *Task* execution policy, the thread processing Receiver task will encounter large overhead due to excessive communication and low computation behavior of Receiver. By assigning *Monitor* execution policy to Receiver this overhead can be reduced.

4.5.3 FileSearch Program from Actor Collections

FileSearch program performs document indexing and searching. The different capsules in this program are: FileCrawler , FileScanner, Indexer and Searcher. FileCrawler recursively visits each sub-directory in the input file path and sends a message to FileScanner whenever it finds a file. FileScanner processes

the file sent by the FileCrawlerand asks next available Indexer from the list of indexers to index the file. Indexer performs hash-based indexing and stores the result. Upon visiting every file in the directories/sub-directories of the input path FileCrawler notifies FileScanner and FileScanner notifies Searcher. Searcher takes search string from the command-line and requests each Indexer to look for the search string in their stored results and return the file path if found.

Capsules *cVector* and the execution policies are shown in the table below. The ACG for this program is shown in Figure 7.

Capsule	cVector	Policy
FileCrawler	<**false**,_,high,high,high>	*Thread*
FileScanner	<**false**,_,high,high,low>	*Task*
Indexer	<**false**,_,low,low,low>	*Monitor*
Searcher	<**true**,_,_,_,_>	*Thread*

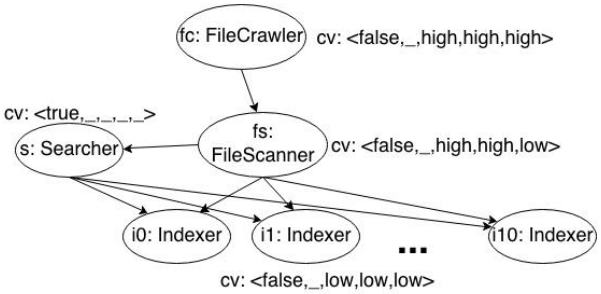

Figure 7. Actor Collection FileSearch

FileCrawler has cVector <**false**,_,high,high,high> because it performs heavy recursive task, hence it is assigned *Thread* execution policy. FileScanner communicates highly with a set of indexers and acts as a delegator. The cVector of FileScanner is <**false**,_,high,high,low> and it is assigned *Task* execution policy. There are eleven Indexer capsule instances with cVector <**false**,_,low,low,low>. These are leaf actors with low computations, hence are assigned *Monitor* policy. Searcher blocks until FileScanner notifies to begin searching. Since it is a blocking capsule, we assigned *Thread* execution policy to it. It is important that FileCrawler is assigned *Thread* execution policy, because it requires a dedicated thread to perform uninterrupted processing. Assigning *Thread* execution policy to blocking Searcher capsule ensures that it does not leads to the starvation of other actors. Our decision to assign *Task* execution policy to FileScanner is critical to further improve the performance of the program. FileScanner in this program can be a performance bottleneck, because it receives large number of requests from FileCrawler and it should delegate the work to Indexer capsules without delaying. Also FileScanner is a stateless capsule and its requests can be simultaneously processed by multiple threads.

4.5.4 BenchErl Serialmsg

BenchErl serialmsg is about message proxying through a dispatcher. The benchmark spawns a certain number of receivers, one dispatcher, and a certain number of generators. The dispatcher forwards the messages that it receives from generators to the appropriate receiver. Each generator sends a number of messages to a specific receiver. This program has 120 instances of Generator capsule, 120 instances of Receiver capsule and one instance of Dispatcher capsule. The table below lists cVector and assigned execution policy for various capsules in this program. The ACG for this program is shown in Figure 8.

Capsule	cVector	Policy
Generator	<**false**,_,high,high,low>	*Task*
Dispatcher	<**false**,_,low,low,low>	*Monitor*
Receiver	<**false**,_,low,low,low>	*Monitor*

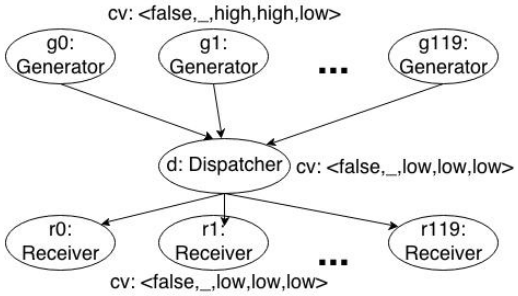

Figure 8. BenchErl Serialmsg

The communication between each Generator with its Receiver is high. Hence, for tightly binding them we assign *Monitor* execution policy to Dispatcher and Receiver so that the thread that is processing Generator will also process Receiver . If Dispatcher is assigned *Task* execution policy, it leads to the performance bottleneck, because it receiver large number of messages from Generator capsules and it should immediately route the messages to appropriate receivers.

5. Evaluation

5.1 Benchmark selection

We have selected actor programs that have data, task, and pipeline parallelism in them at fine and coarse granularity levels. These applications show super-linear, linear and sub-linear speedups and they may or may not scale well when additional cores are allocated to them. Our idea is to evaluate a wide range of actor programs rather than be repetitive. We have selected representative programs from Erlang BenchErl [9] suite, Computer Language Benchmarks Game [3], Actor Collections [4], StreamIt Benchmarks [7], JavaGrande [23] and Panini Examples [1, 20]. While selecting the representative programs from different benchmark

B	Capsule	cVector	Policy
bang	Sender	<false,_,high,high,low>	*Task*
	Receiver	<false,_,low,low,low>	*Monitor*
ehb	Group	<false,_,high,high,low>	*Task*
	Sender	<false,_,high,high,low>	*Task*
	Receiver	<false,_,low,low,low>	*Monitor*
mbrot	Worker	<false,_,high,high,low>	*Task*
	Mandel	<false,_,low,low,low>	*Monitor*
serialmsg	Generator	<false,_,high,high,low>	*Task*
	Dispatcher	<false,_,high,low,low>	*Monitor*
	Receiver	<false,_,low,low,low>	*Monitor*

Figure 9. Details of BenchErl benchmark programs.

suites, we have included programs that consists of different types of actors and their interactions are not straightforward. We have translated a total of fourteen different actor programs to Panini for evaluation. Our rewriting translated only concurrency-related code and structure from the original programs and did not alter or optimize code not related to concurrency. We now list the details about each benchmark program such as capsules, cVectors and execution policies that are assigned to capsules by our mapping technique. The details about the mappings and its impact on the program performance will be discussed in §5.3.

5.1.1 BenchErl Benchmarks

BenchErl is a publicly available scalability benchmark suite for applications written originally in Erlang. Unlike other benchmark suites, which are usually designed to report a particular performance point, BenchErl aims to assess scalability, i.e., a set of performance points that show how an application's performance changes when additional resources (e.g. CPU cores, schedulers, etc.) are added. We have selected bang, ehb, mbrot and serialmsg programs for evaluation. The description of actors (capsules) and the assigned execution policies is shown in Figure 9.

5.1.2 Computer Language Benchmarks Game

This suite of programs is used to compare the performance of different languages and libraries. We have selected fannkuchredux, fasta and knucleotide programs. The description of actors (capsules) and the assigned execution policies is shown in Figure 10.

5.1.3 Programs from Actor Collections

This suite lists a collection of Akka/Scala actor applications in the Github repository. We have selected FileSearch, ScratchPad and PolynomialIntegral actor programs. The description of actors (capsules) and the assigned execution policies is shown in Figure 11.

5.1.4 StreamIt Programs

The set of benchmarks are available with StreamIt software version 2.1.1. Most benchmarks are from the signal processing domain. We have selected BeamFormer and DCT as two representative applications. The description of actors (capsules) and the assigned execution policies is shown in Figure 12.

B	Capsule	cVector	Policy
Fannkuchred	fannkuchred	<false,_,high,low,high>	*Task*
	Collector	<false,_,low,low,low>	*Monitor*
fasta	RandomFasta	<false,_,low,high,high>	*Task*
	RepeatFasta	<false,_,low,high,high>	*Task*
	FloatProbFreq	<false,_,low,low,low>	*Monitor*
	Writer	<false,_,low,low,low>	*Monitor*
Knucleotide	SequenceGen	<false,_,low,low,high>	*Thread*
	Nucleotide	<false,_,low,low,high>	*Task*
	Collector	<false,_,low,low,low>	*Monitor*

Figure 10. Details of CLBG benchmark programs.

B	Capsule	cVector	Policy
FileSearch	FileCrawler	<false,_,high,high,high>	*Thread*
	FileScanner	<false,_,high,high,low>	*Task*
	Indexer	<false,_,low,low,low>	*Monitor*
	Searcher	<true,_,_,_,_>	*Thread*
ScratchPad	FilesystemWalker	<false,_,high,high,high>	*Thread*
	LocAnalyser	<false,_,high,high,low>	*Task*
	LocCounter	<false,_,high,low,high>	*Task*
	Accumulator	<false,_,high,low,low>	*Monitor*
	ResultAccumulator	<false,_,low,low,low>	*Monitor*
Polynomial	DelegateActor	<false,_,low,low,low>	*Monitor*
	DispatcherActor	<false,_,high,high,low>	*Task*
	ComputerActor	<false,_,low,low,low>	*Monitor*

Figure 11. Details of Actor Collections benchmarks.

B	Capsule	cVector	Policy
BeamFormer	AnonFilter_a1	<false,_,high,low,low>	*Monitor*
	AnonFilter_a0	<false,_,high,low,low>	*Monitor*
	InputGenerate	<false,_,high,low,high>	*Task*
	CoarseBeamFirFilter	<false,_,high,low,high>	*Task*
	BeamFirFilter	<false,_,high,low,high>	*Task*
	AnonFilter_a3	<false,_,high,high,low>	*Monitor*
DCT	FileReader	<true,_,_,_,_>	*Thread*
	iDCT8x8_ieee	<false,_,high,low,low>	*Monitor*
	iDCT_2D_reference_fine	<false,_,high,low,low>	*Monitor*
	AnonFilter_a0	<false,_,high,low,low>	*Monitor*
	iDCT_1D_Y_reference_fine	<false,_,high,low,low>	*Monitor*
	YSplitter	<false,_,high,high,low>	*Task*
	iDCT_1D_reference_fine	<false,_,high,low,high>	*Task*
	iDCT_1D_X_reference_fine	<false,_,high,low,low>	*Monitor*
	FileWriter	<false,_,low,low,low>	*Monitor*

Figure 12. Description of various capsules, their respective cVectors and assigned execution policies for Streamit Beamformer and DCT benchmarks.

5.1.5 RayTracer from JavaGrande

This benchmark measures the performance of a 3D ray-tracer. The description of various capsules, their respective cVectors and assigned execution policies are shown in table below.

Capsule	cVector	Policy
Runner	<**false**,_,low,high,low>	*Thread*
RayTracer	<**false**,_,low,low,high>	*Task*

5.1.6 Histogram from Panini Examples

This actor program implements classic histogram problem. The goal of this problem is to count the number of times each ASCII character occurs on a page of text.

Capsule	cVector	Policy
Reader	<**true**,_,_,_,_>	*Thread*
Bucket	<**false**,_,high,low,low>	*Monitor*
Printer	<**false**,_,low,low,low>	*Monitor*

5.2 Methodology

We compare our heuristic-based mapping technique against default thread and default task mapping techniques. The rationale behind comparing against default thread mappings is that most actor languages and frameworks support threaded actors such that programmers can debug/profile their threaded actor program to fine tune the performance. The default task mappings are supported for event-based programming of actors. We measure the runtime of programs for three different mappings when steady-state performance is reached. Following the methodology of Georges *et al.*[14], steady-state performance is reached when the coefficient of variation of the most recent three iteration times of a benchmark fall below 0.02. We compare the execution time for each benchmark program for the three mapping strategies. We also measure the performance of three different mappings on 2, 4, 8, and 12- cores settings (Linux taskset utility is used for altering core settings on 12-core system). The experiments are conducted on 12-core system (2 Six-Core AMD Opteron ® 2431 Processors) with 24GB of memory running the Linux version 3.5.5 and Java version 1.7.0_06. A Java VM size of 2GB is sufficient to run all of our experiments.

5.3 Performance and Scalability

For comparing the performance of our heuristic-based mapping against default-thread and default-task mappings, we define following two metrics:

- I_{th} is the improvement over default-thread mapping. It is the percentage reduction in runtime over default-thread mappings, and
- I_{ta} is the improvement over default-task mapping. It is the percentage reduction in runtime over default-task mappings.

We compute I_{th} and I_{ta} for each program on 2, 4, 8, and 12 core settings. Figure 13 shows the results. We also compute average I_{th} and I_{ta} to determine overall improvement of heuristic-based mapping over default-thread and default-task mapping.

Results. Over fourteen evaluated benchmarks, average I_{th} is 51% and average I_{ta} is 50%. Average I_{th} and average I_{ta} on different core settings is shown in table below.

#cores	I_{th}	I_{ta}
2	49%	50%
4	51%	50%
8	51%	51%
12	53%	50%

Outliers. Mainly for three programs our heuristic-based mapping technique achieved small or no improvements. These programs are, fannkuchredux (I_{th}: 3% & I_{ta}: 1%), RayTracer (I_{th}: -5% & I_{ta}: 10%), and dct (I_{th}: 3% & I_{ta}: 13%). On the other hand, our technique achieved large improvements for five programs. These programs are, bencherl mbrot (I_{th}: 95% & I_{ta}: 95%), bencherl serialmsg (I_{th}: 70% & I_{ta}: 65%), polynomialintegral (I_{th}: 72% & I_{ta}: 72%), fasta (I_{th}: 85% & I_{ta}: 77%), and histogram (I_{th}: 91% & I_{ta}: 99%). We will now discuss these outliers in detail along with other interesting results.

Analysis. For fourteen evaluated benchmarks on average 50% improvement over default mappings suggests that our heuristic-based mapping could be used as a better initial mapping strategy than default-thread and default-task mappings. On average 50% improvement on various core settings indicates that our heuristic-based mapping is consistent across machines with different resource (CPU cores) availabilities.

Our heuristic-based mapping technique achieved small improvements for three programs (**RayTracer**, **fannkuchredux**, and **DCT**). These programs are mainly data-parallel applications with embarrassingly parallel behavior. The results support our earlier intuition that for embarrassingly parallel applications it is easy to determine actors to threads mappings, as actors independently perform the tasks. For instance, in RayTracer program Runner acts as master that distributes the work to a set of RayTracer worker actors. RayTracer worker actors perform independent computations. For this program, mapping is intuitive. Runner could be assigned *Thread* execution policy and each RayTracer worker could be assigned *Task* execution policy. Hence, it is easy to map actors to threads and there is very little opportunity for further improving the initial mapping.

Our heuristic-based mapping technique achieved large improvements for five programs (**mbrot**, **serialmsg**, **polynomialintegral**, **fasta**, **histogram**). These programs have sub-

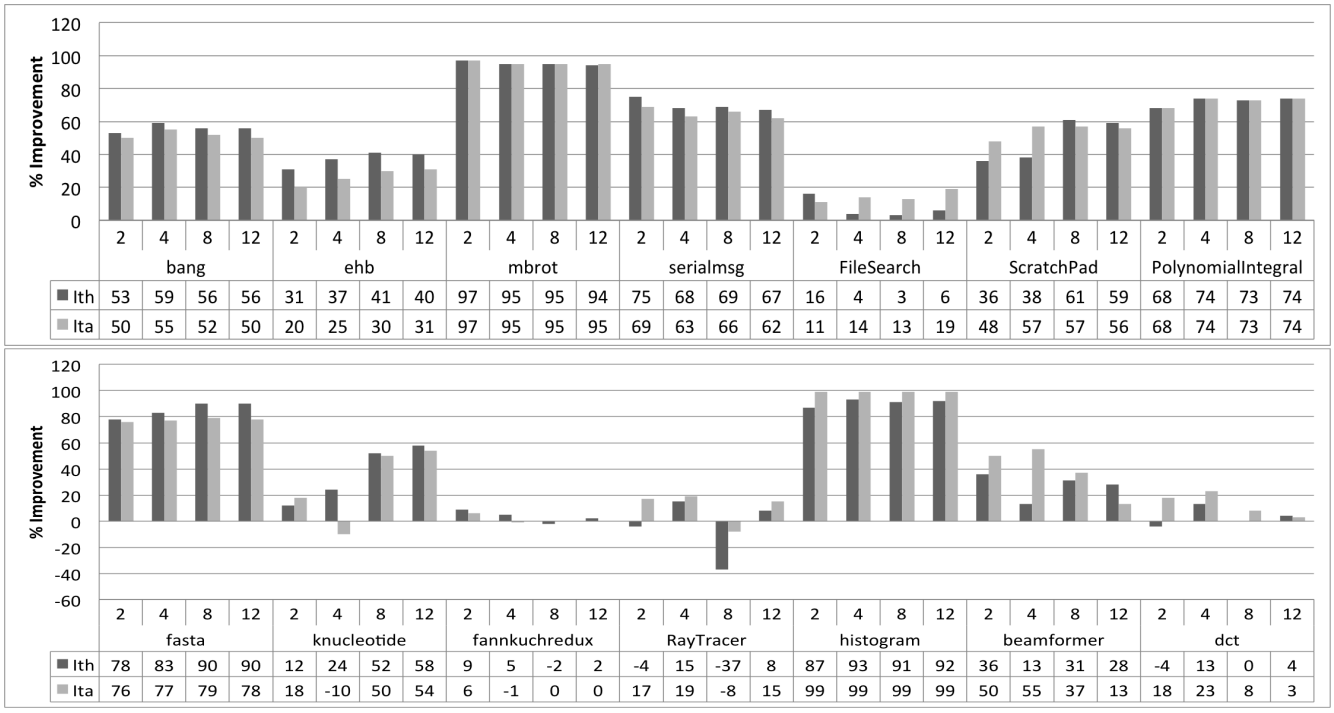

Figure 13. Results show I_{th} (improvement over default-thread mapping) and I_{ta} (improvement over default-task mapping) for the benchmark programs on 2, 4, 8 and 12-core settings.

linear performance benefits. In other words, in these programs balancing computation over communication is a difficult task. **BenchErl-mbrot** program is a Mandelbrot simulator. This program generates a set of pixels that corresponds to a 2-D image of a specific resolution. These pixels are distributed to a set of workers (Worker) which communicates with Madel capsule to check if the pixel belongs to Mandel set or not. By assigning *Monitor* execution policy to Madel capsule large communication overhead between Worker capsules and Madel is reduced. Also, Madel capsule *busy-waits* until the Worker capsules requests Madel to perform Mandel set check. The assignment of *Monitor* execution policy to Madel eliminates this *busy-waits* because the Worker itself performs Madel set check. In **BenchErl-serialmsg** program binding Generator capsules to Receiver capsules by assigning *Monitor* execution policy to Dispatcher large communication overhead between Generator and Receiver is reduced.

In **Polynomial** there are 500 instances of ComputeActor which performs small computations and do not require dedicated thread. By assigning *Monitor* execution policy thread processing DispatchActor processes ComputeActor capsules. In **fasta**, the two instances of RandomFasta capsules communicate highly with their respective FloatProbFreq instances. FloatProbFreq is leaf-capsule and has low computation. By assigning *Monitor* execution policy we eliminate communication overheads. Writer capsule *busy-waits* until complete set of sequences are generated by RandomFasta and RepeatFasta capsules. By assigning *Monitor* execution policy communication overhead is reduced. In **Histogram** program, Reader that has external I/O blocking is assigned *Thread* execution policy. Bucket and Printer perform very small computations. The communication between Reader and 128 instances of Bucket is very large. By assigning *Monitor* execution policy, Reader and Buckets are processed by same thread.

Details. So far we have discussed only outliers and their performance. We will now investigate the remaining benchmark programs to see what mappings were crucial in producing good performance. In **bang**, by assigning *Monitor* execution policy for Receiver, it is now processed by the thread that is processing the Sender. This reduced the communication overhead between Sender and Receiver and improved the performance of the program. Similarly in **ehb** program, by assigning *Monitor* execution policy to Receiver we have reduced communication overhead between Senders and Receivers in each group. In **ScratchPad** program running FileSystemWalker and LocAnalyzer concurrently is important. FileSystemWalker performs high computations recursively and communicates the intermediate results to LocAnalyzer. LocAnalyzer delegates its work to a set of LocCounter instances. Both Accumulator that collects and processes the intermediate results and ResultAccumulator that collects final set of results *busy-waits* until all the requests from FileSystemWalker are processed by LocCounter instances. Hence, assigning *Monitor* execution policies to busy-wait capsules greatly reduces the communication overhead. **knucleotide** has SequenceGen that reads input DNA sequence and uses blocking read opera-

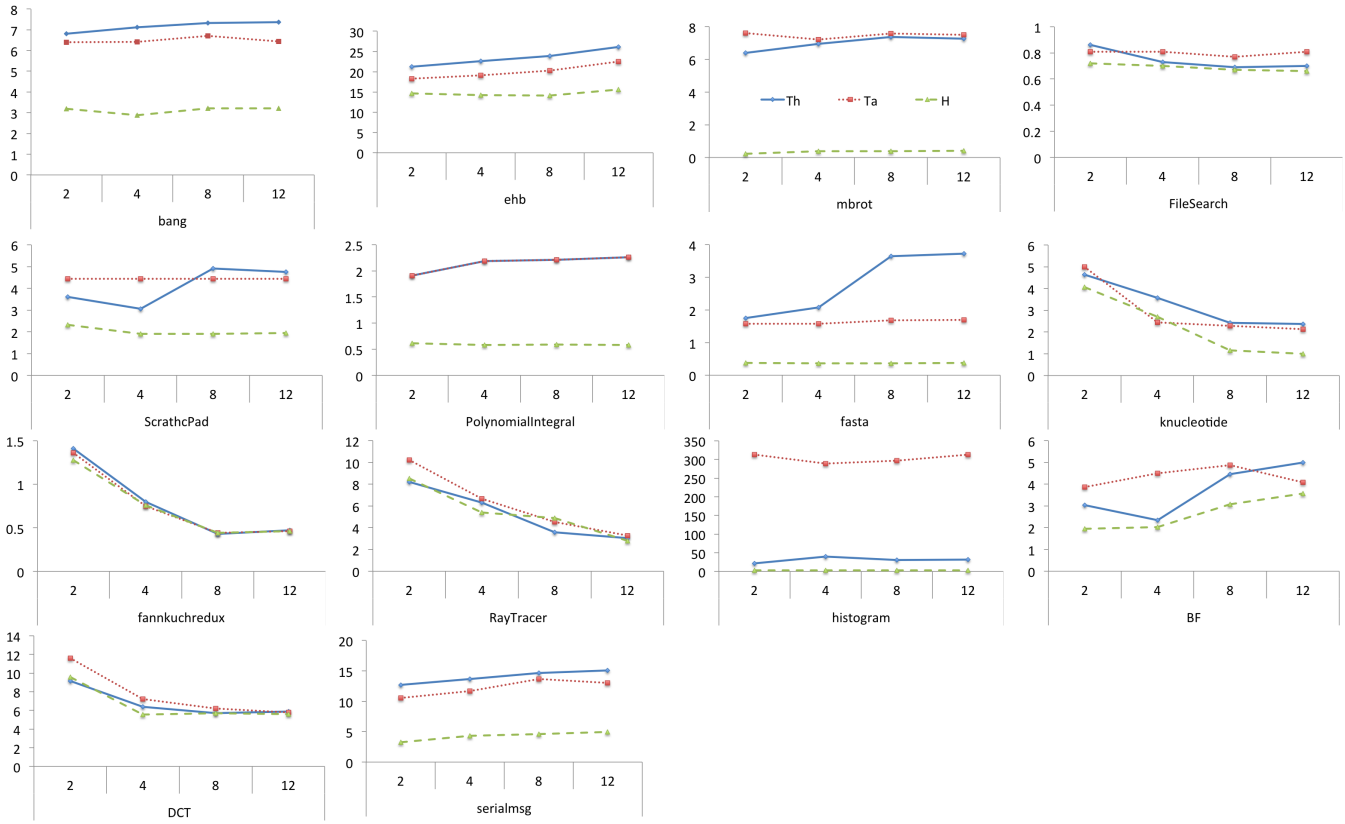

Figure 14. Comparing scalability of our heuristic-based mapping against default-thread and default-task mapping (x-axis: 2, 4, 8, and 12-core settings and y-axis: runtime in seconds).

tion. Hence it is assigned *Thread* execution policy. There are 46 instances of Nucleotide capsule. Nucleotide performs CPU intensive computations and sends the result to Collector capsule. Collector capsule busy-waits until the results from all 46 Nucleotide capsules is received. By assigning *Monitor* execution policy to Collector we eliminate this busy-wait that greatly improves the performance of this program.

In summary, for these programs a heuristic-based mapping such as the one we proposed yields better performance over default mapping techniques. These programs also shows the characteristics that are hard to fine tune for programmers to gain additional performance benefits.

Figure 14 evaluates the scalability of three mapping techniques. The individual charts show the effect of adding cores on program runtime. It can be seen that our heuristic-based mapping technique suffers less from scalability issues when compared to the default mappings. When additional cores are allocated to the program, additional JVM threads are available for the program to use. The default mappings utilize the additional threads without properly adjusting the mappings. In case of highly parallel programs, additional threads balance the workload and helps to improve the overall performance of the program. Whereas, in other program additional threads reduces the computation-to-communication ratio and adds more overhead. This behavior is evident when we profile the default mappings on additional cores we see the number of context-switches rapidly increases. However, in our heuristic-based mapping, when additional threads are available, threads are utilized by keeping the critical mappings intact. For instance, we do not separate hub actors from its affinity actors when additional threads are available for use. This helps us introduce less overheads. To further illustrate, consider fasta program. This program has two instances of RandomFasta, RepeatFasta, one two FloatProbFreq and one Writer. When additional threads are available, it is important that the mapping technique does not assign separate threads for RandomFasta and FloatProbFreq capsules. This is not ensured in default mappings.

Predictive Power of cVector. Our mapping function shown in Figure 4 indicates that *BLK* is the most powerful cVector field. If *BLK* is *true*, we ignore other fields in cVector and assign *Thread* execution policy. In the remaining cases, it is mandatory that other fields in cVector should be checked to assign appropriate execution policy. *STATE* field in cVector is unused for deciding the execution policy. However, this field is used to further enhance the mapping as follows. If any actor is assigned *Task* execution policy and if *STATE* is *false* it indicates that multiple threads can process messages

in the actor's message queue simultaneously. For instance, consider FileSearch program shown in Figure 7, FileScanner capsule has *STATE* value *false*. By allocating more threads to FileScanner task, overall performance of the program can be further improved.

Handling Dynamism. Many actor languages and frameworks allow dynamic creation of actors and actor communication graph may not be available statically. In our mapping technique, when an actor instance is created dynamically, it is assigned an execution policy based on the actor type (or actor definition). Hence, every actor created statically or dynamically is assigned an execution policy. When the actor communication graph cannot be determined statically, we run the program (using default thread mapping) on sample inputs to fully or partially determine actor communication graph and we use the computed graph for mapping actors to threads efficiently. In summary, we believe that the dynamism in actor applications can be accounted in our actors to threads mapping technique.

5.4 Threats to Validity

A threat to validity is our evaluation that uses benchmarks translated to Panini. However, we compare the performance of three different mapping techniques using the same panini program and not three different implementations of the program. The three mapping techniques are implemented in Panini's runtime such that we achieve different actors to threads mapping.

6. Conclusion

In this paper we investigated different aspects of actor-based programs that influence the mapping of actors to JVM threads. We proposed a heuristic-based approach that given an actor program and its communication graph produces actor characteristics vectors for actors. Using the actor characteristics vectors, we apply a set of heuristics in a systematic manner to assign execution policies for actors. The execution policies for actors defines how actors are mapped to JVM threads. We proposed using our heuristic-based mapping in place of default mappings. Our evaluation over a wide range of actor programs shows that our heuristic-based mapping achieves on average 50% improvement over default mappings. Further, our heuristic-based mapping technique assigns execution policies to actors automatically at compile-time. This helps to reduce programmers time and efforts to arrive at initial optimal mappings.

At a higher-level, we believe that better abstractions that enable improved modularity are important for concurrent programming [17–19]. However, a problem with abstractions in practice is that the abstraction boundaries are often breached for performance reasons. By providing an automatic technique for improving mapping of concurrency abstractions to threads, we hope to mimimize such breach of abstraction and improve portability.

Acknowledgments

This work was supported in part by the NSF under grants CCF-08-46059, CCF-11-17937, and CCF-14-23370,.

References

[1] Panini Programming Language. `http://paninij.org/`.

[2] Tim Jansen. Actors guild. `https://code.google.com/p/actorsguildframework/`, 2009.

[3] Computer language benchmarks game. `http://benchmarksgame.alioth.debian.org/`.

[4] Actor Collection. `http://actor-applications.cs.illinois.edu/`.

[5] Mike Rettig. Jetlang. `http://code.google.com/p/jetlang/`, 2008-09.

[6] Panini Benchmark Programs. `design.cs.iastate.edu/strategy/`.

[7] StreamIt. `http://groups.csail.mit.edu/cag/streamit/index.shtml/`.

[8] G. A. Agha. Actors: a model of concurrent computation in distributed systems. 1985.

[9] S. Aronis, N. Papaspyrou, K. Roukounaki, K. Sagonas, Y. Tsiouris, and I. E. Venetis. A scalability benchmark suite for erlang/otp. In *Proceedings of Erlang '12*, pages 33–42.

[10] M. Astley. The actor foundry: A java-based actor programming environment. In *Open Systems Laboratory, University of Illinois at Urbana-Champaign, 1998-99*.

[11] M. Bagherzadeh and H. Rajan. Panini: A concurrent programming model for solving pervasive & oblivious interference. Technical Report 14-09, Iowa State University, August 2014.

[12] T. Desell and C. A. Varela. Salsa lite: A hash-based actor runtime for efficient local concurrency.

[13] E. Francesquini, A. Goldman, and J.-F. Méhaut. Actor scheduling for multicore hierarchical memory platforms. In *Proceedings of Erlang'13 workshop*, pages 51–62. ACM.

[14] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. pages 57–76.

[15] M. Gupta. *Akka Essentials*. Packt Publishing Ltd, 2012.

[16] P. Haller and M. Odersky. Actors that unify threads and events. In *Proceedings of Coordination Models and Languages 2007*.

[17] Y. Long, S. L. Mooney, T. Sondag, and H. Rajan. Implicit invocation meets safe, implicit concurrency. In *GPCE '10: Ninth International Conference on Generative Programming and Component Engineering*, October 2010.

[18] H. Rajan. Building scalable software systems in the multicore era. In *2010 FSE/SDP Workshop on the Future of Software Engineering*, Nov. 2010.

[19] H. Rajan, S. M. Kautz, and W. Rowcliffe. Concurrency by modularity: Design patterns, a case in point. In *2010 Onward! Conference*, October 2010.

[20] H. Rajan, S. M. Kautz, E. Lin, S. L. Mooney, Y. Long, and G. Upadhyaya. Capsule-oriented programming in the Panini language. Technical Report 14-08, 2014.

[21] H. Sasaki, T. Tanimoto, K. Inoue, and H. Nakamura. Scalability-based manycore partitioning. In *Proceedings of the 21st int. Conf. on PACT*, pages 107–116, 2012.

[22] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on multi/many-core systems: survey of current and emerging trends. In *Proceedings of DAC'13*, page 1. ACM.

[23] L. Smith, J. Bull, and J. Obdrizalek. A parallel Java Grande benchmark suite. In *ACM/IEEE Conf. on Supercomputing*, pages 6–6, 2001.

[24] T. Sondag and H. Rajan. Phase-based tuning for better utilization of performance-asymmetric multicore processors. In *International Symposium on Code Generation and Optimization (CGO)*, April 2011.

[25] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In *Proceedings of ECOOP'08*.

[26] A. Tousimojarad and W. Vanderbauwhede. An efficient thread mapping strategy for multiprogramming on manycore processors. *arXiv preprint arXiv:1403.8020*, 2014.

[27] J. Zhang. Characterizing the scalability of erlang vm on many-core processors. 2011.