Abstraction and performance, together at last; Auto-tuning message-passing concurrency on the java virtual machine

by

Ganesha Upadhyaya

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Hridesh Rajan, Major Professor
Gurpur M. Prabhu
Steve Kautz

Iowa State University

Ames, Iowa

2015

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. I would like to thank Dr. Hridesh Rajan for his guidance, patience and support throughout this research and the writing of this thesis. Thanks are due to the US National Science Foundation for financially supporting this project under grants CCF-08-46059, CCF-11-17937 and CCF-14-23370.

I would like to thank my committee members Dr. Gurpur M. Prabhu and Dr. Steve Kautz for their efforts and contributions to this work. Also, I would like to thank the reviewers of AGERE SPLASH 2014 and PLDI 2015 conference for their insightful feedback. I would like to extend my thanks to all the members of Laboratory of Software Design for offering constructive criticism and timely suggestions during research.

I am very grateful to my parents for their moral support and encouragement throughout the duration of my studies.

ABSTRACT

Performance tuning is the leading justification for breaking abstraction boundaries. We target this problem for message passing concurrency (MPC) abstractions on the Java Virtual Machine (JVM). Efficient mapping of MPC abstractions to threads is critical for performance, scalability, and CPU utilization; but tedious and time consuming to perform manually. We solve this problem by putting forth a technique for automatically mapping MPC abstractions to JVM threads. In general, this mapping cannot be found in polynomial time. Our surprising observation is that characteristics of MPC abstractions and their communication patterns can be very revealing, and can help determine the mapping. Our technique addresses a number of challenges that leads to improved performance: i) balancing the computations across JVM threads, ii) reducing the communication overheads, iii) utilizing the information about cache locality, and iv) mapping MPC abstractions to threads in a way that reduces the contention between JVM threads. We have realized our technique in the Panini language that has *capsules* as an MPC abstraction. We also compare our mapping technique against four default mapping techniques: thread-all, round-robin-task-all, random-task-all and work-stealing. Our evaluation on wide range of benchmark programs shows that our mapping technique can improve the performance by 30%-60% over default mapping techniques.

CHAPTER 1. INTRODUCTION

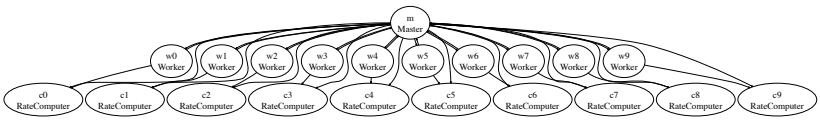
Message-passing based concurrency (MPC) is an approach to concurrency, where there are self-contained concurrently runnable entities that communicate via message passing. Examples of MPC abstractions include: actors[13], active objects [20], guardians [17], capsules [23], etc. A number of MPC frameworks support building large scale distributed applications on the Java Virtual Machine (JVM), e.g. Akka [16], Scala Actors [12], ActorFoundary [5], SALSA [7], Panini [23], etc. We will use the term *abstraction* to mean MPC abstraction except when ambiguous.

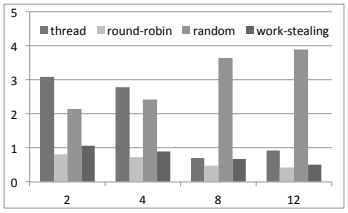
1.1 Mapping Abstractions to Threads

Although, MPC model exposes parallelism by design and the parallelism stems from being able to execute multiple MPC abstractions in parallel, abstractions needs to be mapped to cores carefully for utilizing the multicore. Mapping is a two step process: 1) abstractions to threads mapping and 2) threads to cores mapping (or scheduling). Often, the MPC runtime handles both steps by creating required threads and scheduling them on different cores using an abstraction to core mapping technique.

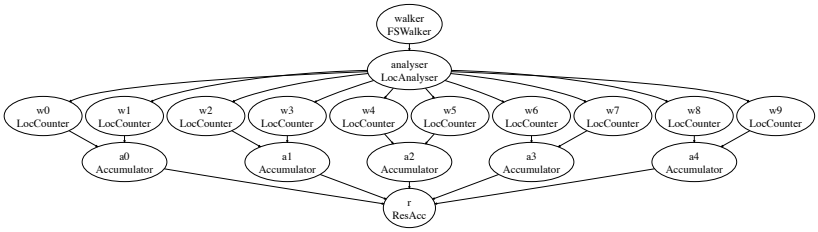
However, in case of MPC frameworks that run on JVM platforms, abstractions to JVM threads mapping is performed by programmers and JVM leaves scheduling of threads on multicore to the OS scheduler. These frameworks provide several kind of schedulers and dispatchers to programmers using which they can map the abstractions in their applications to JVM threads, e.g. Akka has four kind of dispatchers, Scala has two kind of schedulers, Panini has four kinds of dispatchers, etc. The availability of wide-variety of schedulers and dispatchers suggests that it is important to map MPC abstractions to JVM threads carefully to utilize multicore efficiently.

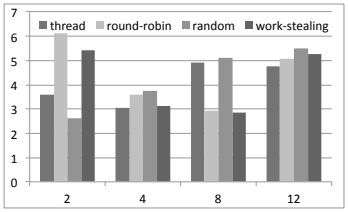
A large number of discussions on tuning MPC abstractions (or tuning schedulers/dispatchers) [3] indicate that programmers find it hard to manually arrive at the optimal mapping. For example, a





LogisticMap *from Savina [14]: computes the Logistic Map [19] using a recurrence relation.*





ScratchPad *application from Actor Collections [33]: counts lines for all files in a directory.*

Figure 1.1 Communication graphs and program running time comparison charts for LogisticMap (logmap) and ScratchPad benchmark programs. x-axis:2, 4, 8, and 12 core settings and y-axis: runtime (in seconds). Lower bars are better.

Stackoverflow user asks: "*We're using Akka in such a way where we have two separate dispatchers ... We're now running into some performance issues and we're looking into how we can tune the dispatcher configuration parameters and see how they affect the performance of the application [http://tinyurl.com/ohwsesn]*". In response an expert user replies: "*I strongly encourage you to monitor your system to find the root cause of your performance issues and not just randomly tweaking the configuration"* Another Stackoverflow expert advises: "*Tuning the performance of an actor system is a hard task. But without knowing what the task is (e.g. does it do any blocking operations, read from IO, ...) it is hard to tell what to do or if anything can be done at all... [http://tinyurl.com/ovypwnu]*"

When manual tuning is hard, programmers use the default mappings (default schedulers/dispatchers) 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[1], improving the performance is tedious and time consuming [28].

Moreover, a single default mapping strategy may not work across programs. To illustrate consider this problem for two programs shown in Figure 1.1. We investigate the performance of four widely used default mappings: i) thread, ii) round-robin, iii) random and iv) work-stealing. In thread mapping,

[1]These are concurrent applications that are not designed with parallelism in mind. They often shows degradation in the performance upon adding more resources such as, cores

every abstraction is assigned a dedicated thread and in other three mappings abstractions are tasks that are assigned to taskpool or threadpool. Figure 1.1 shows the performance (program runtime) for these default mappings over 2, 4, 8 and 12 core settings. For LogisticMap program, round-robin task based mapping out-performs other three mappings, whereas for ScratchPad program, there is no clear winner. On 2-core setting, random task based mapping does better, on 4, and 12-core settings thread based mapping does better and on 8-core setting work-stealing task based mapping does better. These results illustrate that single default mapping strategy may not work across programs.

When manual tuning is hard and default mappings may not produce the desired performance, a brute force technique that tries all possible combinations of abstractions to threads mapping (using different kinds of schedulers/dispatchers) could be used. However, this approach suffers from combinatorial explosion. Even for an MPC program with few kinds of concurrent entities, the number of combinations that must be tried is large. For instance, for an MPC program with eight kinds of concurrent entities, trying all possible combinations of four kinds of schedulers/dispatchers requires exploring 65536 (4^8) different configurations. In such situations, finding a mapping solution that yields significant performance improvement over default mappings is desirable.

Our key observation is that, *local computation and communication behavior* of a concurrent entity in a MPC program is surprisingly predictive for determining *globally beneficial* mapping to threads. Here, by computation and communication behaviors we mean properties such as: externally blocking behaviors, local state, computational workload, message send/receive pattern, and inherent parallelism. A related observation is that determining these behavior at a coarse/abstract level is sufficient to solve this important problem.

1.2 Contributions

Our work makes several contributions: (1) We propose *characteristics vector* (cVector), a representation for computation and communication behavior of an MPC abstraction. Main challenges in coming up with this representation strategy were to select suitable fields and then to formulate cVector in a *language-agnostic* manner. (2) We describe analyses for determining components of a *characteristics vector*. Main challenge here was in coming up with local analyses, which can be performed on

single abstraction at a time — especially challenging for communication patterns. (3) We describe an automatic cVector to thread mapping technique, which relies on several novel intuitions. (4) We implement this technique in the Panini compiler to map capsules, an MPC abstraction, to threads. (5) We examine applicability of our approach to other MPC abstractions.

For evaluating our approach, we have selected four default mapping strategies (that are representative schedulers/dispatchers in this domain). We profile program execution time and cpu consumption and use them as metrics to compare our mapping technique against four default mapping strategies. Our results show 30%-60% improvement in program execution times when compared to default mappings. Since our technique is automatic, it could help decrease the efforts required for performance tuning.

1.3 Thesis Outline

The rest of this thesis is organized as follows. First, we give background on Panini capsules. In Chapter 3 we describe cVector, cVector analyses, and cVector to thread mapping technique. In Chapter 4 we describe our evaluation. In Chapter 5, we compare and contrast with related ideas, and Chapter 6 concludes.

CHAPTER 2. BACKGROUND

2.1 Panini Capsules, an MPC Abstraction

A capsule is an MPC abstraction implemented in the programming language Panini [2, 23, 6, 22]. Figure 2.1 presents an example HelloWorld program in this language. In this program there are three capsules HelloWorld, Greeter and Console and they are connected as *HelloWorld* $\rightarrow$ *Greeter* $\rightarrow$ *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 2.1 HelloWorld Program in Panini

In Panini's programming model, capsules are independently acting entities. Capsules provide interfaces 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.

Capsules internally use message-passing based concurrency mechanism to process inter-capsule procedure calls. A capsule contains a message queue for receiving messages, a thread for processing messages, a set of state variables that represents its local state and a message processing logic containing set of message handlers (mapped to capsule procedures). Capsules cannot share data, multiple threads cannot process capsule's messages simultaneously, and capsules can have finite number of nested capsules.

CHAPTER 3. MAPPING MPC ABSTRACTIONS TO THREADS

In this chapter we describe our technique for mapping MPC abstractions to threads starting with our representation strategy. We explain our technical innovations using capsules as an example and then revisit their applicability to other MPC abstractions.

3.1 Selecting Computation and Communication Features

Performance tuning of MPC applications involves balancing the computations (performed by various entities) across available resources and reducing the communication overheads. Selecting computation and communication features to analyze is the first step toward that goal with four challenges: selected feature must be representative, amenable to detection by analyses, provide sufficient coverage, and avoid duplication. We now describe our selected set of features. Note that selecting other features is also possible, but for our purpose the following suffice.

3.1.1 Blocking behavior

We believe that it is vital to account for blocking behavior that can arise due to I/O, socket or database blocking primitives. When a concurrent entity externally blocks, it not only adds additional overhead to its computation, it may also lead to starvation of other concurrent entities in the system (thread processing this entity may not be available to process messages from other entites when they share a thread). For instance, consider the code snippet from *Searcher* capsule in *FileSearch* Panini program shown below. This code snippet reads an input string from console and uses this word to query *Indexer* capsules, which returns the file paths containing the search word. Such externally blocking behaviors, blocks the thread processing the message.

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.print("Enter the word you want to search");
String word = br.readLine(); // externally blocks the thread
```

```
1 Map<Integer,Integer> dataMap;
2 void write(DictionaryConfig.WriteMessage
      writeMessage) {
3   ...
4   dataMap.put(writeMessage.key, writeMessage.
        value);
5   ...
6 }
```

```
1 void read(DictionaryConfig.ReadMessage
      readMessage) {
2   ...
3   if (dataMap.get(readMessage.key) != null)
4     value = dataMap.get(readMessage.key);
5   ...
6 }
```

3.1.2 Local state

We also consider the state of a concurrent entity, i.e. the set of state variables as an important feature. State variables may be of primitive data types or large objects such as collections and hash-maps. The message handlers of a MPC abstraction may read or write to its local state. When a abstraction has a large local state, the thread processing abstraction's message can benefit if abstraction's local state is available in its cache while processing the subsequent messages (improves cache locality). For instance, consider the code snippet of *Dictionary* capsule from *concdict* Panini program. This code snippet contains local state *dataMap* and two message handlers *write* and *read*. Both message handlers read/write to large local state *dataMap*. Subsequent read/write requests can benefit, if *dataMap* is available in the local cache of the thread processing the messages of *Dictionary* capsule.

3.1.3 Computational workload

Understanding the nature of computations performed by the MPC abstraction is also important. MPC abstractions may perform CPU intensive computations. MPC abstractions that don't perform CPU intensive computations could share a thread resulting in overall saving of resources.

3.1.4 Communication pattern (or message send/receive pattern)

An MPC abstraction may send messages to multiple recepients or receive messages from multiple senders. Message send and receive patterns can be used to place senders and recepients that commu-

```
1 Sender capsule:
2   Receiver receiver;
3   void send(String msg) {
4     for (int i=0; i<440; i++)
5       receiver.receive(msg);
6   }
```

```
1 Receiver capsule:
2   void receive(String msg) {
3     System.out.println("MSG Recv!"+msg);
4   }
```

```
1   DispatcherCapsule server;
2   double compute(ComputationContext context) {
3     double val = server.computeAreaUnderTheCurve(context);//Future to store result
4     return val; // claiming the result from Future
5   }
```

nicate often. Therefore, knowing the pattern of send and receive communications is important. For instance, consider the code snippet of *Sender* and *Receiver* capsules from *bang* Panini program shown below. *Sender* capsule has one-to-many send pattern (for every message it receives, it sends 440 messages to *Receiver* capsule). Since *Sender* communicates with *Receiver* often, by mapping *Sender* and *Receiver* to the same thread, message processing overheads could be greatly reduced.

3.1.5 Inherent parallelism

MPC abstractions during their computation may communicate with other concurrent entities and may require results from them to continue its computation. *Inherent parallelism* represents the nature of parallelism that can be exploited during this synchronization. Consider following code snippet from *DelegateCapsule* that communicates with *DispatcherCapsule* in message handler *compute*. The message handler defines a *Future*[1] (line 3) when the message is sent to *DispatcherCapsule*. The result returned from *DispatcherCapsule* is used right away (line 4). Here the inherent parallelism is zero. *Inherent parallelism* could be used to decide if two communicating capsules can be assigned dedicated threads to benefit from parallelism.

[1] *Future* is a sort of a placeholder object that an capsule can create for a result that does not yet exist. The result of the *Future* is computed concurrently and can be later collected.

3.2 Characteristics Vector (cVector): Representing Computation and Communication Behaviors of MPC Abstractions

We define characteristic vector (cVector) to represent the computation and communication behaviors described in §3.1. The cVector has five fields: $\langle \beta, \sigma, \pi, \rho, \omega \rangle$. The key challenge is to assign coarse/abstract values to cVector fields (that represents behaviors of an MPC abstraction) which can be determined using local program analysis (program analysis on the MPC abstraction). For instance, for deciding capsules to threads mapping, every capsule type is assigned a cVector and cVector fields are determined by analyzing the capsule type. The rest of this section describes cVector fields, domain of values that can be assigned to them and the local program analysis steps to determine them for *Panini Capsules*.

Blocking. β represents *blocking behavior* of capsules that use externally blocking primitives such as I/O, socket and database blocking primitives and it can be assigned values $true$ or $false$. A capsule with blocking behavior may block the executing thread, may lead to starvation of other capsules and system deadlock. For determining capsules with blocking behaviors the local program analysis analyzes message handlers of capsules and identifies the usage of blocking library calls such as *InputStream.read()*, *ServerSocket.accept()* etc. We assign $\beta = true$, if any of the capsule procedures use blocking library calls, i.e. this is an intra-capsule analysis.

Local State. σ represents *local state* of capsules. State of a capsule includes the parameters sent during the creation of the capsule and the local state variables defined as part of the capsule. $\sigma \in \mathscr{S} = \{nil, primitive, large\}$ are the legal values and σ is determined as follows:

- nil, if no state variables
- $primitive$, when state variables are of primitive data types
- $large$, when state variables use large data structures such as objects, collections, maps etc.

The larger the local state of a capsule, the higher the probability of a pre-fetch fail (cache miss). To determine σ we check the type of each state variable of the capsule, which is also an intra-capsule analysis.

Inherent Parallelism. π represents *inherent parallelism* exposed by the capsule when it communicates (sends a message) with other capsules. The communication between capsules is asynchronous, however capsules may receive results from other capsules and use the result immediately or later. Based on this we assign following values: $\pi \in \mathscr{P} = \{sync, async, future\}$ and values are determined as follows:

- *sync*, if a capsule sends a message to another capsule and waits for the result, which is consumed immediately
- *async*, if a capsule sends a message and does not require the result
- $future$, otherwise (includes cases where a capsule receives result to a *Future*[1] and uses it later on).

Inherent parallelism (π) is defined for each message handler of a capsule and we define $\vec{\pi}$ to represent the inherent parallelism of a capsule (one entry for each message handler). We analyze each message handler at every communication point (location of sending message) and assign values to π as described above. In case of multiple communication points in a message handler, we count *sync*, *async* and $future$ values and assign whichever dominates. If none dominates, we assign π = $future$ for the message handler. Inherent parallelism of the capsule is then assigned a value from $\{sync, async, future\}$ such that the value dominates message handlers. If none dominates, we assign π = $future$ for the capsule. So, finding inherent parallelism is also an intra-capsule analysis.

For example, consider the code snippet of *SeriesWorker* capsule from LogisticMap Panini program shown below.

```
1 void nextTerm() {
2   curTerm = computer.compute(new ComputeMessage(senderId, curTerm));
3 }
4 void getTerm() {
5   master.process(new ResultMessage(curTerm));
6 }
```

It has two message handlers, *nextTerm* and *getTerm* with one communication point each. For message handler *nextTerm*, π = *sync* and for message handler *getTerm*, π = *async*. We assign $\pi = future$ for *SeriesWorker* capsule, since none of $\{sync, async\}$ dominates.

Communication Pattern. ρ represents the *communication pattern* of the capsule ($\rho \in \mathscr{R} = \vec{\mathscr{R}} \times \overleftarrow{\mathscr{R}}$). A communication pattern is a tuple ($\vec{\rho}$, $\overleftarrow{\rho}$) consisting of message send pattern ($\vec{\rho} \in \vec{\mathscr{R}}$) and message receive pattern ($\overleftarrow{\rho} \in \overleftarrow{\mathscr{R}}$). The message send pattern can be assigned following values { $leaf$, $router$, $scatter$ } and the values are determined as follows:

- $leaf$, does not send messages to other capsules or just replies to sender
- $router$, sends exactly one message to the another capsule
- $scatter$, sends more than one message to one or more capsules.

Some capsules communicates with a set of capsules more often than others (applicable to abstractions in any MPC model). These capsules form hub-affinity groups. To determine if a capsule (hub) has higher affinity to a set of capsules (affinity capsules), we use its message send pattern, $\vec{\rho}$. The idea is that by making the thread that processes messages of the hub capsule also process the messages of its affinity capsules. This helps to reduce the communication overheads between hub and affinity capsules. If a capsule is assigned $scatter$ value then it indicates that the capsule forms hub-affinity group with capsules that it communicates with. When a capsule is assigned $router$ or $leaf$ for $\vec{\rho}$, that capsule may be part of the affinity group of some other capsule.

The message receive pattern, $\overleftarrow{\rho} \in \overleftarrow{\mathscr{R}}$ can be assigned following values $\{gather, request\text{-}reply\}$ and the values are determined as follows:

- *gather*, if a capsule is expected to receive large number of messages
- *request-reply*, if a capsule is expected to receive small number of messages.

The local program analysis used to determine communication patterns ($\vec{\rho}$, $\overleftarrow{\rho}$) is described in §3.3.

Computational Workload ω represents capsules *computational workload*. We classify the computations performed in message handlers to be CPU intensive or not. Each message handler is assigned a legal value from $\mathscr{W} = \{math, io\}$ and the value is assigned as follows:

- *math*, indicates cpu intensive computations, when message handler has recursive functions, loops with unknown bounds, makes high cost library calls, uses heavy data structures such as objects, collections, maps, receive large data and read/write to capsule state that is heavy
- *io*, no cpu intensive computations.

Upon computing $\vec{\omega}$, which has an entry for each message handler, the computational workload of the capsule is assigned computational workload of the message handler that has message receive pattern, $\overleftarrow{\rho} = gather$. If none of the message handlers have message receive pattern, $\overleftarrow{\rho} = gather$ then ω for the capsule is assigned *math*, if there exists a message handler with $\omega = math$, otherwise ω is assigned *io* for the capsule.

3.3 Local Program Analysis to Determine Communication Pattern

Generally, the topology is required to determine message send and receive patterns ($\vec{\rho}$, $\overleftarrow{\rho}$). The key challenge is to determine them without relying on the topology. That is, by performing local program analysis of the abstraction. We propose a technique that predicts message send and receive patterns by analyzing how message handlers are defined. Figure 3.1 shows a number of message handler patterns and the assigned values for ρ. We now describe each of them in detail. In the description we use a term "connected capsules" to refer to the set of capsules that a capsule can send message to.

The pattern *send(capsules[i])* indicates that the capsule can send messages to its connected capsules, but it sends message to only one of its connected capsules. We can derive from this behavior that the capsule is going to receive more messages that performs *send(capsules[i])* to send messages to all its connected capsules. Here, the message sending pattern is one-to-one, hence we take $\vec{\rho}$ to be *router*. We have predicted that this message handler of the capsule is going to receive many messages, hence predict $\overleftarrow{\rho}$ to be *gather*.

The pattern *state_rw + {send(capsule) or cond(send(capsule))}* indicates that the capsule read/write its local state and sends message to its connected capsule (always or on condition). Here, the message sending pattern is one-to-one, hence $\vec{\rho}$ is assigned *router*. The message handler pattern indicates that all functionality of the capsule is performed in this message handler (read/write state, sending message

Message Handler Pattern	**Description**	$\overrightarrow{\rho}$	$\overleftarrow{\rho}$
send(capsules[i])	sends message to one of its capsules	*router*	*gather*
state_rw + {send(capsule) or cond(send(capsule))}	read/write state and sends message to connected capsule (always/on-condition)	*router*	*gather*
send_all(capsules)	sends message to all connected capsules	*scatter*	*request-reply*
cond(send_all(capsules))	sends message to all connected capsules on condition	*router*	*gather*
exit_calls	contains library calls that indicates termination such as exit(), print() etc.	N/A	*request-reply*
partial_rw	read/write state partially	N/A	*gather*

Figure 3.1 Predicting $\overrightarrow{\rho}$ and $\overleftarrow{\rho}$ using message handler patterns.

to its connected capsules), hence we predict that this message handler is often executed and we assign value *gather* to $\overleftarrow{\rho}$.

The pattern *send_all(capsules)* indicates that the capsule sends message to all its connected capsules. Hence, $\overrightarrow{\rho}$ has value *scatter*. A behavior such as, broadcasts to all is less likely to happen often in MPC because broadcasting of messages can lead to congestion in the system or system not responding nature. Hence, we predict that such message handlers will be executed less and we assign *request-reply* value to $\overleftarrow{\rho}$, which indicates small message receive frequency.

The pattern *cond(send_all(capsules))* indicates that the message handler conditionally performs broadcasting of the message. $\overrightarrow{\rho}$ for such a message handler will be *router*. We predict that the capsule receives many messages, performs state update and when some condition on its state variable is satisfied it broadcasts the message. Hence, we assign *gather* to $\overleftarrow{\rho}$, which indicates high message receive frequency.

The last two message handler patterns does not apply to predict $\overrightarrow{\rho}$ and they only predict $\overleftarrow{\rho}$. In the pattern *exit_calls*, if the message handler uses library calls that indicates possible termination (for instance, exit(), print() etc), we predict that such messages will be received less. We assign *request-reply* value to $\overleftarrow{\rho}$. In *partial_rw* pattern, we predict that capsule that reads or writes to its local state partially will receive many such messages to perform complete read/write to its local state before the end of its

existence. We assign *gather* to $\overleftarrow{\rho}$, which indicates high message receive frequency. §3.6 provides an example, where we show how the local program analysis described here is applied to compute cVectors for capsules in a Panini program.

3.4 Execution Policies for Abstractions

We formulate the problem of mapping capsules to threads as assigning execution policies to capsules in the program. Execution policy defines how capsule messages are processed. We focus on following four execution policies,

- *THREAD*, in this execution policy, a dedicated thread is assigned for processing the messages from the capsule's message queue and executing the corresponding behavior (works similar to Akka's pinned dispatcher).
- *TASK*, in this execution policy, the capsule messages are processed by the shared thread of the taskpool. The taskpool may contain one or more capsules that abide to *TASK* execution policy. The order in which the messages from different capsules message queue has to be processed could vary. One simple policy is to process one message from each capsule to avoid starvation of other capsules.
- *SEQ/MONITOR*, in case of *SEQ* and *MONITOR*, the policy is that the capsule that sends the message needs to execute the defined behavior at the capsule that received the message (works similar to Akka's calling thread dispatcher).

It can be seen that, by assigning different execution policies to capsules, they get mapped to threads differently. For instance, assigning thread execution policy leads to one-to-one capsules to threads mapping. Assigning task execution policy leads to n-to-one capsules to threads mapping. The principle behind selecting these four execution policy is that, they represent the mapping constructs (or dispatcher/schedulers) available in the widely used JVM-based MPC frameworks. Note that, the execution policies described here are applicable to any MPC abstraction.

3.5 Mapping Heuristics: Deciding Abstractions to Threads Mapping

So far we have described cVector as a way to represent MPC abstraction's computation and communication behaviors. In this section, we examine a number of heuristics that assigns execution policies to capsules with certain cVectors.

Blocking Heuristics. This heuristic states that a capsule that has externally blocking behavior (β = $true$), should be assigned thread (Th) execution policy. Assigning any other policy to blocking capsules may block the executing thread, may lead to starvation of other capsules and system deadlock.

Heavy Heuristics. This heuristic states that a capsule that is non-blocking, communicates often with other capsules and performs CPU intensive computations should be assigned thread (Th) execution policy. The rationale behind this decision is that the dedicated thread can perform its CPU intensive computations in parallel with other threads without voluntarily interruptions (meaning voluntarily not giving up CPU to other threads). The cVector of such a capsule is: <$false$, nil, •, $scatter$, •, $math$>. Note that, • as value for any cVector field indicates that any value from the domain of values can be assigned and the value does not influence the mapping decision.

HighCPU Heuristics. a capsule that is non-blocking, communicating less often with other capsules and performing CPU intensive computations should be assigned task (Ta) execution policy. By assigning task execution policy, the capsule can share a thread with other capsules (which are also assigned task policy) and can also benefit from work-stealing optimizations available for the task/thread pools. The cVector of such capsules are: <$false$, •, •, $router$, •, $math$>.

LowCPU Heuristics. This heuristics states that a capsule that has cVector like <$false$, •, •, $leaf$, •, io> should be assigned monitor (M) execution policy. Such a capsule does not need a thread for processing its messages and the threads of the capsules that sends the message themselves will process the messages of the capsule.

Hub Heuristics. This heuristic states that hub capsules should be assigned task (Ta) execution policy. Hub capsules are represented using cVector <$false$, nil, •, $scatter$, •, io>. The rationale behind this decision is that affinity capsules (capsules that a hub capsule communicates with) can be executed by the shared thread that is processing the messages of the hub capsule.

Affinity Heuristics. This heuristic states that affinity capsules should be assigned monitor (M) execution policy. Affinity capsules have following cVector <*false*, •, •, *router*, *request-reply*, *io*>. By assigning monitor execution policy, the thread processing the hub capsule (of these affinity capsules) will process the messages of the affinity capsules.

Master Heuristics. There are two types of master capsules. First type has cVector <*false*, *primitive/large*, •, *scatter*, •, *io*>. This type of master capsules sends messages to worker capsules and may not receive reply from workers. Second type has cVector <*false*, •, •, *router*, *gather*, *io*>. This type of master capsules does not send messages to worker capsules, however they receive messages from worker capsules. The heuristics states that both types of master capsules must be assigned task (Ta) execution policy. Master capsules have the property that they delegate the work to worker capsules.

Worker Heuristics. This heuristic states that worker capsules should be assigned task (Ta) execution policy. The cVector of worker capsules is <*false*, •, •, *leaf*, •, *math*>. Similar to HighCPU capsules, these capsules can utilize any load-balancing strategies applied to the task/thread pool.

3.6 Mapping Function

Using the heuristics stated above, we have defined a mapping function as shown in Figure 3.2. The mapping function takes capsule cVector as input and assigns an execution policy. By following the flow it is easy to see that the function is complete. Because, every capsule with a cVector is assigned an execution policy. It can also be seen that capsules are never assigned multiple execution policies.

We now present an example where we construct cVectors for capsules and apply our mapping function to determine the execution policies for capsules. Consider the capsule communication graph (topology) of LogisticMap program from Savina [14] shown in Figure 1.1. This Panini program has three types of capsules: *Master*, *SeriesWorker* (10 instances) and *RateComputer* (10 instances). *Master* communicates with each *SeriesWorker* (each *SeriesWorker* replies back to *Master*) and *RateComputer*. *SeriesWorker* instances also communicates with *RateComputer*. One can see that the communication graph of LogisticMap Panini program is not simple and determining the execution policies to capsules (or capsules to threads mapping) for such a program is non-trivial. For capsules in LogisticMap Panini program, we construct cVectors and determine the execution policies using our technique.

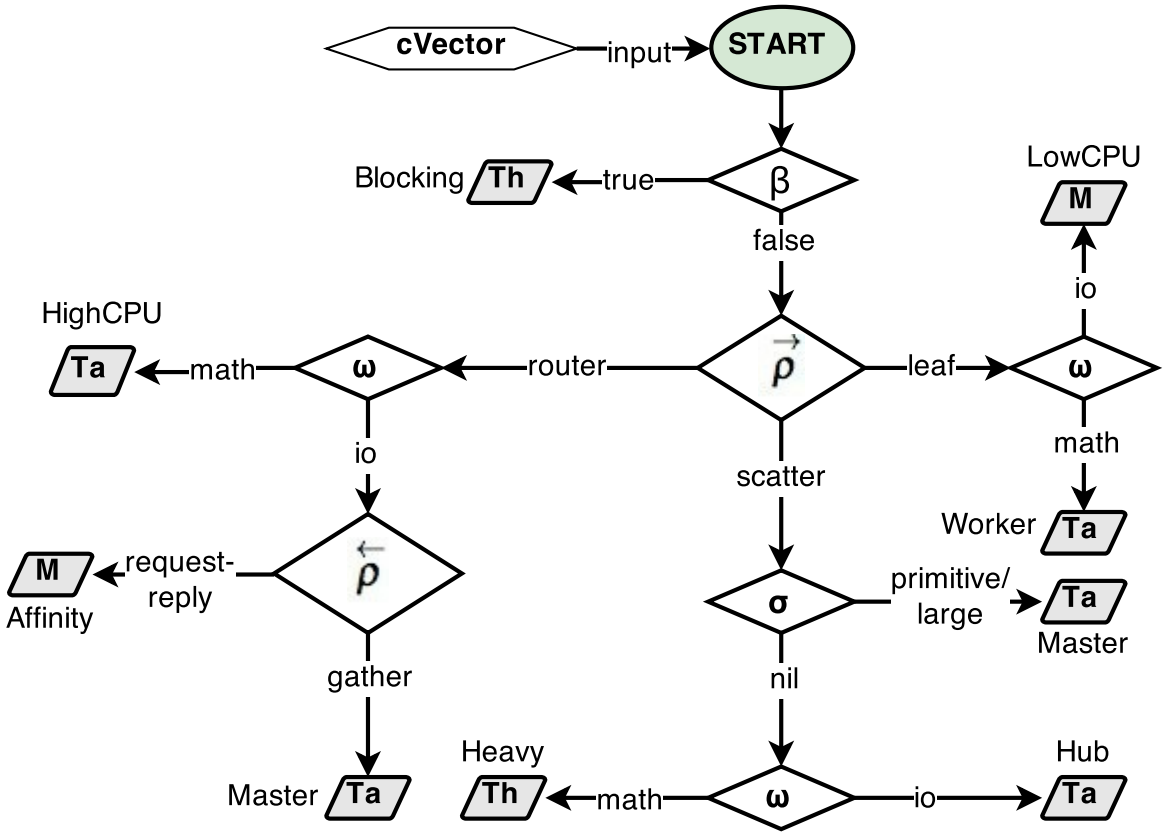


Figure 3.2 Flow diagram of our mapping function.

```
1  void begin(LogisticMapConfig.StartMessage sm) {
2    int i = 0;
3    while (i < LogisticMapConfig.numTerms) {
4      for (SeriesWorker worker : workers) {
5        worker.nextTerm();
6      }
7      i += 1;
8    }
9    for (SeriesWorker worker : workers) {
10     worker.getTerm();
11     numWorkRequested += 1;
12   }
13 }
```

```
1  void process(LogisticMapConfig.ResultMessage rm)
       {
2    termsSum += rm.term;
3    numWorkReceived += 1;
4    if (numWorkRequested == numWorkReceived) {
5      System.out.println("Terms sum: " + termsSum);
6      for (SeriesWorker worker : workers) {
7        worker.done();
8      }
9      for (RateComputer computer : computers) {
10       computer.done();
11     }
12   }
13 }
```

Figure 3.3 LogisticMap Master capsule code snippet. For brevity, we show only required code. The source code of complete program can be found in [1].

Master. This capsule does not use blocking calls, hence $\beta = false$ and it has local state defined using primitive data types, hence $\sigma = primitive$. Figure 3.3 shows two message handlers *begin* and

process. The message handler, *begin* uses *send_all(capsules)* pattern (lines 4-6 and lines 9-12), hence $\overrightarrow{\rho}$ = *scatter* and $\overleftarrow{\rho}$ = *request-reply* (as shown in *send_all(capsules)* rule shown in Figure 3.1). The message handler, *process* has pattern *cond(send_all(capsules))* (lines 4-12) and uses *print()* (line 5) library call that indicates possible termination, hence $\overleftarrow{\rho}$ = *request-reply* and $\overrightarrow{\rho}$ = *router*. Given, ($\overrightarrow{\rho}$, $\overleftarrow{\rho}$) = (*scatter*, *request-reply*) for message handler *begin* and ($\overrightarrow{\rho}$, $\overleftarrow{\rho}$) = (*router*, *request-reply*) for message handler *process*, ($\overrightarrow{\rho}$, $\overleftarrow{\rho}$) for *Master* will be (*scatter*, *request-reply*). The inherent parallelism, π = *async*, as all message sends are asynchronous in message handlers and ω = *io*, as there is no computation intensive operations in either of the message handlers. Hence, cVector for *Master* capsule is, <β, σ, π, $\overrightarrow{\rho}$, $\overleftarrow{\rho}$, ω> = <*false*, *primitive*, *async*, *scatter*, *request-reply*, *io*> and by following the mapping function (shown in Figure 3.2) for this cVector gives us task (Ta) execution policy.

```
1  double startTerm = 0;
2  double curTerm = 0;
3  void nextTerm() {
4    int senderId = id;
5    curTerm = computer.compute(new
6      ComputeMessage(senderId, curTerm));
7  }
8  void getTerm() {
9    master.process(new ResultMessage(curTerm));
10 }
11 void done() {
12   exit();
13 }
```

```
1  double rate = 0.0;
2  ResultMessage compute(ComputeMessage
       computeMessage) {
3    double result = computeNextTerm(
         computeMessage.term, rate);
4    int senderId = computeMessage.senderId;
5    return new ResultMessage(result);
6  }
7  void done() {
8    exit();
9  }
```

Figure 3.4 LogisticMap SeriesWorker capsule (on left) and RateComputer capsule (on right) code snippets.

SeriesWorker. This capsule is non-blocking and has local states defined using primitive data types (lines 1-2), hence β = *false* and σ = *primitive*. Figure 3.4 (left) shows three message handlers, *nextTerm*, *getTerm* and *done*. The message handlers, *nextTerm* and *getTerm* both have pattern *state_rw and send(capsule)* pattern (lines 5-6 and line 9), hence ($\overrightarrow{\rho}$, $\overleftarrow{\rho}$) = (*router*, *gather*). The message handler, *done* has *exit()* (line 12), hence $\overleftarrow{\rho}$ = *request-reply*. Given, ($\overrightarrow{\rho}$, $\overleftarrow{\rho}$) for all three message handlers as above, ($\overrightarrow{\rho}$, $\overleftarrow{\rho}$) = (*router*, *gather*) for *SeriesWorker*. None of the message handlers have CPU intensive code, hence ω = *io*. The message handler, *nextTerm* uses the returned result immediately, hence π = *sync*. The message handler, *getTerm* uses asynchronous send, hence π = *async*. For *SeriesWorker*, π = *future* as none of {*sync*, *async*} dominates. Hence, cVector for *SeriesWorker* capsule is, <β, σ, π, $\overrightarrow{\rho}$,

$\overleftarrow{\rho}$, ω> = <$false$, *primitive*, *sync*, *router*, *gather*, *io*> and execution policy for this cVector is task (Ta) execution policy.

RateComputer. This capsule is non-blocking and has local state defined using primitive data types, hence $\pi = false$ and σ = *primitive*. Figure 3.4 (right) shows two message handlers, *compute* and *done*. The message handler, *compute* has pattern *state_rw and send(capsule)*, hence ($\overrightarrow{\rho}$, $\overleftarrow{\rho}$) = (*router*, *gather*). The message handler, *done* has *exit()* call, hence $\overleftarrow{\rho}$ = *request-reply*. Given that, ($\overrightarrow{\rho}$, $\overleftarrow{\rho}$) = (*router*, *gather*) for *RateComputer*. Since this capsule replies to sender capsule, $\overrightarrow{\rho}$ should be changed to $leaf$. For this capsule, π = *async* and ω = *io*. Hence, cVector is, <β, σ, π, $\overrightarrow{\rho}$, $\overleftarrow{\rho}$, ω> = <$false$, *primitive*, *async*, $leaf$, *gather*, *io*> and execution policy for this cVector is monitor (M) execution policy.

3.7 cVector+

Upon deciding the execution policies for capsules, we show that the mappings can be further improved for a specific case. To perform this analysis we use capsule communication graph (CCG) and execution policies of capsules. An CCG is a directed graph G(V,E) where, $V = A_0, A_1, ..., A_n$ is a set of nodes, each representing a capsule, and 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. An example CCG is shown in Figure 1.1. This specific case can be described as follows: there exists a capsule with monitor (M) execution policy and it communicates with a set of parent capsules (capsules that sends message to this capsule) that have task (Ta) execution policies, these parent capsules have ω = *io*, $\overrightarrow{\rho}$ = *router* and $\overleftarrow{\rho}$ = *request-reply*. In such a case, the parent capsules will be part of a taskpool that is served by a set of threads (size = #cores). The solution we propose is to cut-down the size of the taskpool by half (size = #cores/2). This will improve the program runtime due to reduced number of lock contentions between threads that are processing parent capsules when trying to communicate with a capsule that is assigned monitor execution policy. It also helps to reduce the CPU consumption of the program by increasing the workload on threads (more tasks per thread). In our benchmark suite, we have improved both program runtime and cpu consumptions for six Panini programs that exhibits this special case using cVector+ mapping. One such Panini program is bang. This program contains a number of *Sender* capsules that are assigned task execution policy and they all communicates with a single *Receiver* capsule which has monitor policy assigned to it.

3.8 Applicability to Other JVM-based MPC Frameworks

In general, the proposed technique is applicable to other JVM-based MPC frameworks. In the proposed technique, we have selected abstraction behaviors that are commonly seen in MPC frameworks. We have represented the abstraction behaviors as cVectors, the local program analysis for determining coarse/abstract values to cVectors only uses the abstraction (does not rely on the topology), the execution policies described are available in most MPC frameworks and the heuristics that maps abstraction cVectors to execution policies are based on the intuitions of general MPC abstractions.

Most MPC frameworks have dynamism in terms of creating new abstractions or creating new communications dynamically. Our cVector based mapping technique assigns execution policy to abstraction types, hence dynamically created abstraction instances just inherit the execution policy assigned to its abstraction type.

CHAPTER 4. EVALUATION

4.1 Benchmarks

We have implemented our technique in Panini Capsules [2, 23, 6] and is available from [1]. For evaluating our technique, we have selected representative programs from Erlang BenchErl Suite [4], Actor Collections [33], Computer Language Benchmarks Game [9], JavaGrande [30], StreamIt Benchmarks [34], and Savina Actor Benchmarks [14]. The representative benchmark programs are concurrent or parallel applications, which exhibits different concurrency patterns and parallelisms (data, task, pipeline). These applications show super-linear, linear and sub-linear speedups and they may not scale well when additional cores are allocated to them. Our idea is to evaluate a wide range of programs rather than be repetitive. While selecting the representative programs from different benchmark suites, we have included programs that consists of abstractions with different behaviors and their interactions are not straightforward. We have translated a total of fifteen representative programs to Panini [23, 2] for evaluation. Panini translations of these programs have one-to-one correspondence with the source language program (meaning, capsules in the translated Panini program is exactly same as the MPC abstractions in the source program). For instance, Scala Actors (as MPC Abstractions) in logmap benchmark from Savina and Capsules (as MPC Abstractions) in Panini version are exactly same and they perform same computations and communications. Figure 4.1 lists our Panini translated benchmark programs and it also shows the distribution of execution policies. The assignment of execution policy and its impact on the program performance is discussed in §4.3.

Suite	Benchmark	Policy Distribution			
		Thread	Task	Monitor	Sequential
BenchErl	bang	0	1	1	0
	mbrot	0	1	1	0
	serialmsg	0	1	1	1
Actor Collections	FileSearch	2	1	0	1
	ScratchPad	1	2	2	0
	Polynomial	2	1	0	0
CLBG	fasta	0	2	1	1
	Knucleotide	1	1	1	0
	Fannkuchredux	1	0	1	0
JG	RayTracer	1	1	0	0
Streamit	BeamFormer	0	3	2	2
	DCT	1	1	2	4
Savina	logmap	1	1	1	0
	concdict	0	2	1	0
	concsll	0	2	1	0

Figure 4.1 Lists Panini translated benchmark programs [1] and it shows the distribution of execution policies of capsules.

4.2 Methodology

We compare our cVector based mapping technique against four widely used mapping techniques in JVM-based MPC frameworks: 1) *thread*, 2) *round-robin*, 3) *random* and 4) *work-stealing* (hereon, we refer to these mappings as default mappings). In *thread* mapping, each abstraction (capsule, actor, etc) is assigned a dedicated JVM thread. In *round-robin* mapping, a collection of abstractions are served by a pool of threads in round-robin manner. In *random* mapping, a collection of abstractions are served by a pool of threads at random, i.e. a thread picks an abstraction to serve at random. In *work-stealing* mapping, abstractions are assigned to threads but these threads can steal work from other abstractions, if idle. We have implemented these four default mappings in Panini Capsules and our comparison uses the same Panini program.

We measure program *runtime* and *CPU consumption* for *thread*, *round-robin*, *random*, *work-stealing* and our *cVector* mappings when the steady-state performance is reached. Following the methodology of Georges *et al.*[10], the 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 program *runtime* and *CPU consumption* for these five 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 max heap size of 2GB is sufficient to run all of our experiments.

4.3 Performance Evaluation

For comparing the performance of our cVector based mapping against default mappings, we define I_{th}, I_{rr}, I_r and I_{ws} as percentage reduction in program *runtime* over *thread*, *round-robin*, *random* and *work-stealing* mappings respectively. Note that, we reuse these metrics for comparing the percentage reduction in program *CPU consumption* over *thread*, *round-robin*, *random* and *work-stealing* mappings.

We compute I_{th}, I_{rr}, I_r and I_{ws} for each benchmark program for *runtime* and *CPU consumption* on 2, 4, 8, and 12 core settings. We also compute average I_{th}, I_{rr}, I_r and I_{ws} to determine overall performance improvement of our cVector based mapping over default mappings for program *runtime* and *CPU consumption*.

Results. Figure 4.2 shows I_{th}, I_{rr}, I_r and I_{ws} for both program *runtime* and *CPU consumption* for our benchmark programs. Overall, our cVector based mapping technique showed 40.56%, 30.71%, 59.50%, and 40.03% improvements over *thread round-robin*, *random* and *work-stealing* mappings respectively (Range: 30% to 60%). We also saw -21.48%, 4.78%, -12.30%, 14.58% changes in cpu consumption (Range: -21% to 15%). These results suggests that our mapping technique does not boost the program runtime by just consuming more resources such as CPU.

Analysis. We now analyze the performance improvements of cVector based mapping for each benchmark program to dig deeper and understand the factors that improved the program runtime and CPU consumption. As seen in Figure 4.2, there are number of interesting cases like improvements in both runtime as well as CPU consumption. For deeper analysis, we profiled the program execution (using *perf*) and collected values for following metrics:

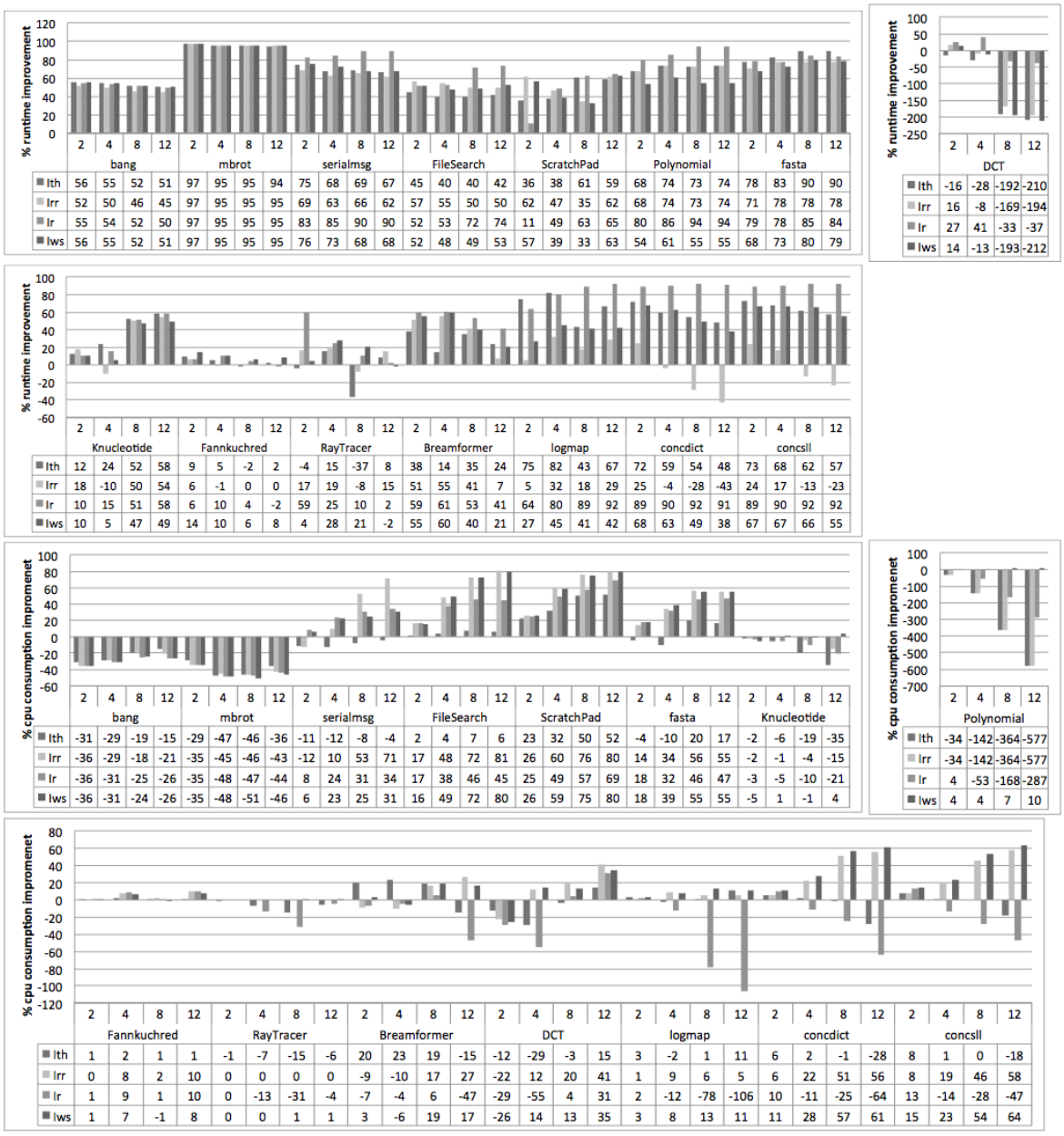


Figure 4.2 First two rows (3 charts) show % runtime improvement over default mappings, next two rows (3 charts) show % cpu consumption improvement over default mappings for fifteen benchmarks. For each benchmark there are four core settings (2, 4, 8, 12-cores) and for each core setting there are four bars (Ith, Irr, Ir, Iws) showing improvement over four default mappings (*thread*, *round-robin*, *random*, *work-stealing*). Higher bars are better.

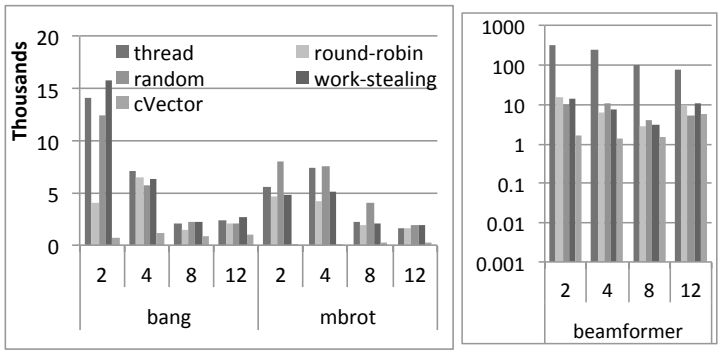


Figure 4.3 Shows reduction in #context-switches in our cVector based mapping when compared to default mappings for *bang*, *mbrot* and *BeamFormer* benchmarks (last bar represents cVector based mapping). Lower bars are better.

- *voluntary context-switches*
- *involuntary context-switches*
- *#cache-miss*, number of cache-miss (cache load/store requests that could not be served by any level cache).
- *#L1-miss*, L1-dcache-load-misses.
- *#LLC-miss*, LLC-load-misses.

Measuring context-switches helps us quantify lock contentions, and measuring *#cache-miss*, *#L1-miss* and *#LLC-miss* helps us quantify cache locality. We now describe major categories in our results.

1) Reduced lock contentions. For Panini benchmarks *bang*, *mbrot*, *BeamFormer* and *Polynomial*, the improvement in performance appears due to higher CPU consumption. However, in case of *bang* and *mbrot*, the program *runtime* is improved also due to the reduction in lock contentions between threads processing various capsules in the program and reduced message processing overheads. In *bang* program, there exists a large amount of lock contentions due to interaction (a large number of messages passed) between *Sender* capsules and a single *Receiver* capsule. Our mapping technique assigned monitor (*M*) execution policy to *Receiver* capsule, which greatly reduced the lock contentions and overheads due to message processing logic. Benchmarks *mbrot* and *BeamFormer* similar reasons for the performance improvements (Figure 4.3).

2) Reduced message-passing overheads. For Panini benchmarks *FileSearch*, *concdict* and *concsll*, the improvement in performance is due to reduced message passing overheads. For example, *concdict* simulates concurrent dictionary access. Capsules in this program are *Master*, *Worker* (20 instances) and *Dictionary*. The concurrently running *Worker* capsules performs 180152 dictionary reads and 19848 dictionary writes. Our technique assigns thread (*Th*) policy to *Master*, task (*Ta*) policy to *Worker* and monitor (*M*) policy to *Dictionary*. By assigning monitor policy to *Dictionary*, *Worker* capsules process 200000 messages themselves. This reduces the message processing overhead substantially (when compared to *Th* or *Ta* policy to *Dictionary*).

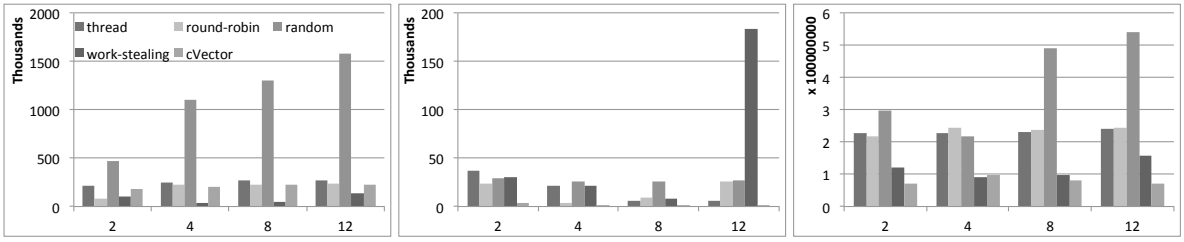


Figure 4.4 Shows reduction in *voluntary context-switches*, *involuntary context-switches* and *#cache-miss* for serialmsg benchmark in our cVector based mapping when compared to default mappings. Lower bars are better.

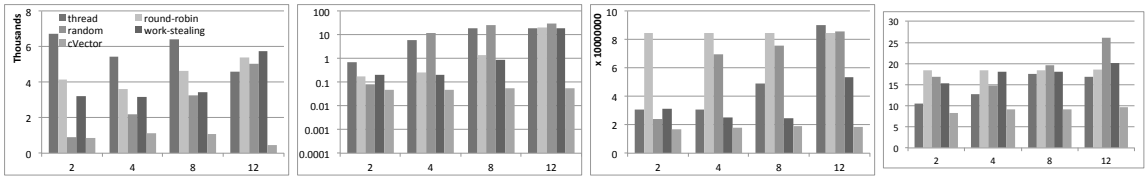


Figure 4.5 Shows reduction in *voluntary context-switches*, *involuntary context-switches*, *#cache-miss* and *#LLC-miss* for ScratchPad benchmark in our cVector based mapping when compared to default mappings. Lower bars are better.

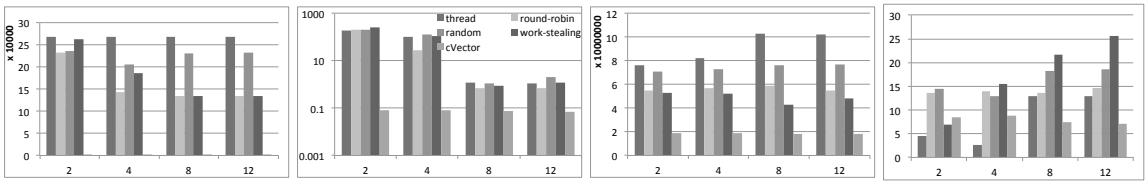


Figure 4.6 Shows reduction in *voluntary context-switches*, *involuntary context-switches*, *#cache-miss* and *#LLC-miss* for fasta benchmark in our cVector based mapping when compared to default mappings. Lower bars are better.

3) Reduced lock contentions and cache-misses. In Panini benchmarks *serialmsg*, *ScratchPad*, *fasta*, the improvement in performance is due to reduction in lock contentions and improved cache locality.

serialmsg. BenchErl *serialmsg* is about message proxying through a dispatcher. The benchmark spawns 120 instances of *Receiver* capsule, one *Dispatcher* capsule, and 120 instances of *Generator* capsule. 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. Our mapping technique assigned task (*Ta*) policy to *Generator* and monitor (*M*) policy to *Dispatcher* and *Receiver* capsules. This allows binding of every *Generator* instance to its respective *Receiver* instance. A large number of messages are sent and received between these capsules, hence reducing the lock contention overheads due to message-processing logic improved the performance immensely. Figure 4.4 shows reduction in #context-switches and *#cache-miss* which supports our hypothesis.

ScratchPad. This Panini program computes line of count for files in the input directory. Capsules in this program are *FSWalker*, *LocAnalyser*, *LocCounter* (20 instances), *Accumulator* (10 instances) and *ResultAcc*. *FSWalker* browses all files in the input directory and sends a message to *LocAnalyser* with file names. *LocAnalyser* sends a message to one of the available *LocCounter* capsules to perform line counting. *Accumulator* and *ResultAcc* collects results from *LocCounter* capsules. For achieving good performance, *FSWalker* and *LocAnalyser* must have good latency (messages are processed as soon as they are received) and *Accumulator* and *ResultAcc* must not become performance bottlenecks. This is achieved in our cVector based mapping technique, which assigns thread (*Th*) policy to *FSWalker* and task (*Ta*) policy to *LocAnalyser*. This makes capsules *FSWalker* and *LocAnalyser* process messages in uninterrupted manner (as *FSWalker*'s dedicated thread can send messages to the taskpool thread of the *LocAnalyser*). Assigning monitor (*M*) policy to *Accumulator* and *ResultAcc* reduces their communication overheads with *LocCounter* capsules. Figure 4.5 shows reduction in #context-switches, *#cache-miss* and *#LLC-miss* which supports our hypothesis.

fasta. This Panini program generates and writes random DNA sequences. Capsules in this program are *RandomFasta* (2 instances), *RepeatFasta*, *FloatProbFreq* (3 instances) and *Writer*. Both *RandomFasta* and *RepeatFasta* capsules independently generates sequence with the help of their respective

FloatProbFreq capsules and sends message to *Writer* capsule for printing. Here, the communication of *RandomFasta* and *RepeatFasta* capsules with their respective *FloatProbFreq* capsules is too large. Our technique assigns sequential (*S*) policy to *FloatProbFreq* capsules to reduce this overhead. Figure 4.6 shows reduction in #context-switches and *#cache-miss* which supports our hypothesis.

Our 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 abstractions to threads mappings, as abstractions independently perform their tasks. For instance, in RayTracer program *Runner* acts as master that distributes the work to a set of *RayTracer* worker capsules. *RayTracer* worker capsules 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 capsules to threads and there is very little opportunity for further improving the mapping.

4.4 cVector+: Further enhancing cVector based mapping

We see a number of benchmark programs that could benefit from our enhanced cVector (cVector+) mapping strategies (described in §3.7). The programs are *bang*, *mbrot*, *serialmsg*, *beamformer*, *concdict* and *concsll*.

Methodology. For these programs, we re-run the experiments with our cVector based mapping but reduced the size of the taskpool to half (initial size of the taskpool was #cores, we reduced it to #cores/2).

Results. Figure 4.8 compares the improvements of cVector+ against cVector mapping strategies and Figure 4.7 provides details about the improvements for each of the six programs. Figure 4.8 shows that, with proposed enhancement to our cVector based mapping, we are able to further reduce program *runtime* for six programs and a substantial improvements can be seen with respect to program *CPU consumptions*. This happens due to reduced context-switches, reduced cache-misses and reduction in cpu consumption as less number of threads are operating. The results supports the fact that reducing number of threads (whenever necessary) reduces program runtime and cpu consumptions, however

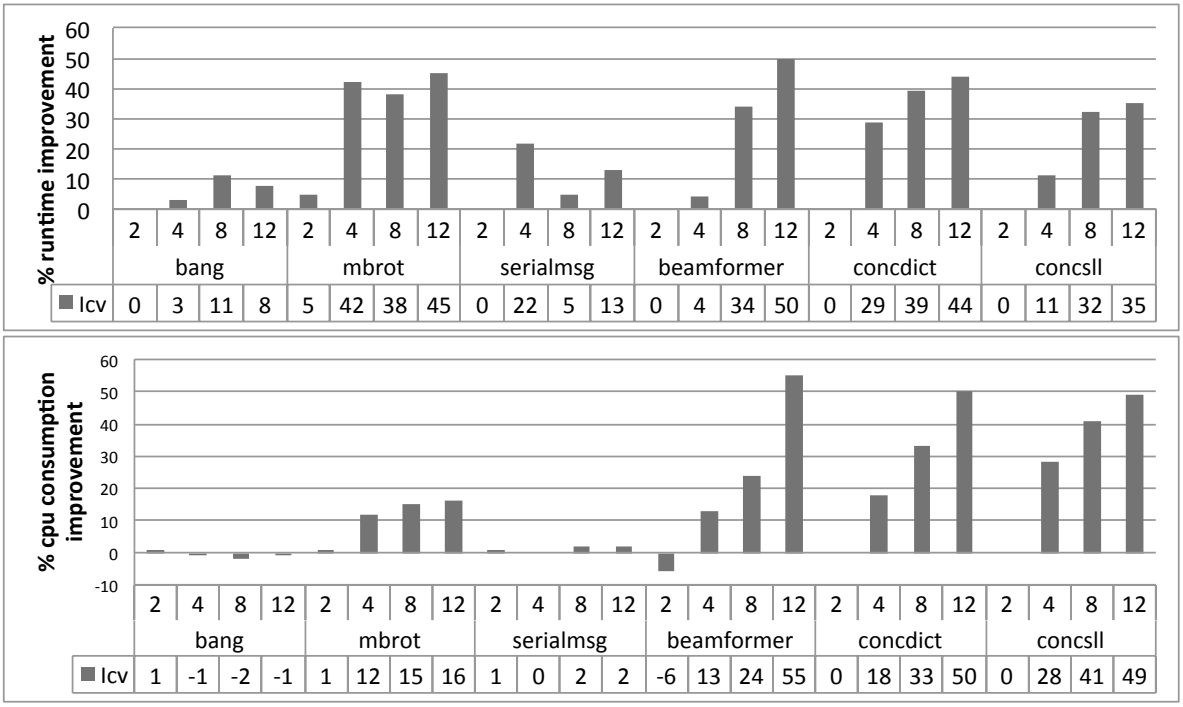


Figure 4.7 Improvements in program runtime and cpu consumptions over cVector based mapping for six programs measured on 2, 4, 8 and 12 core settings (higher is better). Overall improvements of cVector+ over cVector is shown in Figure 4.8

Metric	**cVector**				**cVector+**			
	I_{th}	I_{rr}	I_r	I_{ws}	I_{th}	I_{rr}	I_r	I_{ws}
runtime	40.56	30.71	59.50	40.03	43.51	36.26	60.93	43.00
cpu consumption	-21.48	-4.78	-12.30	14.58	-14.93	-0.65	-4.42	18.61

Figure 4.8 Compares average improvements in program runtime and cpu consumptions for cVector and enhanced cVector (cVector+) mappings.

determining which programs can benefit is the key. We determine this using abstractions cVectors, execution policies and topology.

4.5 Threats to Validity

A threat to validity of our evaluation is that we may not be able to extrapolate our findings to programs that have completely different characteristics compared to our benchmarks. To mitigate this threat, we have selected a wide variety of benchmarks from varied sources.

Second threat to validity of our evaluation is that it is Panini centric (because, i] default mappings are implemented in Panini, and ii] source programs are translated to Panini) and we may not be able to extrapolate our findings to other JVM-based MPC frameworks. To mitigate this threat we have used a representative set of default mappings that are available in most of the widely used JVM-based MPC frameworks. And, Panini programs used in the evaluation have one-to-one correspondence with the source language program (meaning, capsules in the translated Panini program is exactly same as abstractions in the source program). For instance, actors in logmap benchmark from Savina and capsules in its Panini version are exactly same and they perform same computations and communications.

Third threat to validity of our evaluation is that it compares cVector based mappings that use flexible mappings (abstractions can be assigned different execution policies) against default mappings that use single mapping for everything (abstractions are assigned single execution policy). There exists no automatic technique that assigns flexible mappings like ours. In the current state of the art, programmers use default mappings (or default scheduler/dispatcher). They customize the mappings when the performance is poor (which may involve mixing default mappings). We agree that, a comparison against the best manually tuned program would help to strengthen the contribution of our cVector based mapping. However, we could not find such a manually tuned program in the benchmark suites that we have used.

Fourth threat to validity of our evaluation is that we have used a multicore (6+6 core) for evaluation, which may not have the same platform characteristics as a distributed cluster. However, we believe that our results would still be applicable, e.g. for a number of cases we reduce the message passing overhead, which would be specially significant for distributed cluster.

Finally, we have compared only with one candidate from round-robin, random, and work-stealing algorithms, but our selection is a representative from each class of these scheduling algorithms. Future work can explore other variations.

CHAPTER 5. RELATED WORK

5.1 JVM-based MPC Frameworks

Frameworks such as Akka [16], Kilim [32], Scala Actors [12], Jetlang [25], ActorFoundry [5], SALSA [7] and Actors Guild [15] allow programmers to map their actors to JVM threads and fine tune their application using schedulers and dispatchers. The default mappings evaluated in this paper represents these schedulers and dispatchers. Akka provide four kinds of dispatchers: default, pinned, balancing, and calling thread. The default dispatcher is used if programmer does not specify (this is similar to our random mapping strategy). In Kilim, actors are runnable tasks which are assigned to a thread-pool and the scheduling policy is round-robin. Scala Actors allow creation of thread-based and event-based (uses task-pool and round-robin scheduling policy) actors. SALSA allows creation of heavy-weight and light-weight actors using Stage and by default maps actors to a set of stages (can be considered as thread) using round-robin policy. Likewise, other actor frameworks use default actors to threads mapping or programmer specified mappings (using schedulers/dispatchers). When compared to these works, our technique automatically assigns capsules to threads.

5.2 Non-JVM MPC Frameworks

Several works on performance improvement of non-JVM MPC frameworks exists. Francesquini *et al.*[8] proposes a technique implemented in Erlang [37] 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 characteristics along with hub-and-affinity.

5.3 Mapping Task Graphs

Mapping application on to multi-core is a 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 [29] 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 [11, 26, 27, 31]. These techniques are not directly applicable to JVM-based MPC frameworks, because threads to cores mapping is left to OS scheduler and only MPC abstraction to threads mapping can be optimized. However, abstraction to threads mapping technique can utilize solutions proposed for general task graph mapping problem. In our cVector based mapping technique, we utilize characteristics and interaction behaviors similar to task characteristics and task graph in general task graph mapping problem.

5.4 Mapping Problem in Multi-threaded Programs

Note that the mapping problem in MPC 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 MPC programs, the mapping problem is two-fold: mapping MPC abstractions to threads and scheduling of threads on multi-cores. Tousimojarad and Vanderbauwhede[35] propose efficient strategies for mapping threads to cores for OpenMP multi-threaded programs. When compared to this work, our technique maps capsules to threads and not threads to cores. Threads to cores mapping is handled by OS scheduler in JVM-based MPC frameworks.

A preliminary version of this work was presented in our AGERE 2014 workshop paper [36]. We enhanced the work in multiple ways. We incorporate more details to cVector and we assign values to each capsule procedure not just capsule and propagate it to capsule. This enhancement helped us to improve the mappings. We have redefined the predictive power of cVector fields and upgraded the mapping function. In this enhanced work, predicting the incoming message pattern holds the key. We

show that, it can be used to reveal more properties about capsules such as contention and cache locality. We have considered four standard default mapping strategies to compare against instead of two (thread and round-robin-task). Finally, our enhanced cVector based mapping further improved the program execution time and cpu consumptions. The enhanced work is under submission in OOPSLA 2015 conference.

CHAPTER 6. CONCLUSION

Performance optimization is one of the leading reasons for breaking abstraction boundaries. In this work we targeted this problem for message-passing abstractions on JVM, where performance concerns may lead to deformed designs. We proposed a technique to automatically map such abstractions to JVM threads using capsules as a specific use case. We assign execution policies to capsules to achieve capsules to threads mapping. Our mapping technique utilizes a set of properties about capsules and their communications to select execution policies. We have evaluated our mapping technique against four commonly used default mapping techniques in JVM-based MPC frameworks. We have evaluated on a wide-variety of MPC benchmarks that are both concurrent and parallel applications exhibiting different concurrency patterns and parallelisms (data, task, pipeline). Our results show 30%-60% improvement in program execution times when compared to default mappings.

At a higher-level, we believe that better abstractions that enable improved modularity are important for concurrent programming [21, 18, 24]. 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 minimize such breach of abstraction and improve portability. Our mapping technique does not require any changes to the design of the application; it mainly defines how capsules will be compiled, preserving design while improving performance. Our concrete implementation is not applicable to other MPC abstractions, however, other frameworks for JVM can benefit from our technical innovations namely: (1) cVector, (2) cVector analysis, and (3) cVector-to-thread mapping.

BIBLIOGRAPHY

[1] Download link: cVector-based mapping. http://tinyurl.com/ohrzcjh.

[2] Panini Programming Language. http://paninij.org/.

[3] Tuning Dispatchers: Discussion threads. http://tinyurl.com/o6utwkp,http://tinyurl.com/ovypwnu,http://tinyurl.com/ponekm6,http://tinyurl.com/o7xzh8s,http://tinyurl.com/ohwsesn.

[4] S. Aronis, N. Papaspyrou, K. Roukounaki, K. Sagonas, Y. Tsiouris, and I. E. Venetis. A Scalability Benchmark Suite for Erlang/OTP. In *Erlang'12 Workshop*.

[5] M. Astley. The Actor Foundry: A Java-based Actor Programming Environment. In *Open Systems Laboratory, University of Illinois at Urbana-Champaign, 1998-99*.

[6] M. Bagherzadeh and H. Rajan. Panini: A Concurrent Programming Model for Solving Pervasive and Oblivious Interference. In *Modularity'15: 14th International Conference on Modularity*, March 2015.

[7] T. Desell and C. A. Varela. SALSA Lite: A Hash-Based Actor Runtime for Efficient Local Concurrency. In *Concurrent Objects and Beyond, Lecture Notes in Computer Science, 2014*.

[8] E. Francesquini, A. Goldman, and J.-F. Méhaut. Actor Scheduling for Multicore Hierarchical Memory Platforms. In *Erlang'13 Workshop*.

[9] B. Fulgham and I. Gouy. Computer language benchmarks game. http://benchmarksgame.alioth.debian.org/.

[10] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *OOPSLA'07*.

[11] F. Guirado, A. Ripoll, C. Roig, and E. Luque. Performance prediction using an application-oriented mapping tool. In *Proceedings of 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2004*.

[12] P. Haller and M. Odersky. Actors That Unify Threads and Events. In *ICCML COORDINATION'07*.

[13] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI'73*.

[14] S. Imam and V. Sarkar. Savina-An Actor Benchmark Suite. In *AGERE'14 Workshop*.

[15] T. Jansen. Actors guild. https://code.google.com/p/actorsguildframework/.

[16] B. Jonas. Akka, TypeSafe Inc. http://akka.io.

[17] B. Liskov and R. Scheifler. Guardians and Actions: Linguistic support for robust, distributed programs. *TOPLAS '83*, 5.

[18] 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.

[19] R. May. Simple mathematical models with very complicated dynamics. In *The Theory of Chaotic Attractors, 2004*.

[20] O. M. Nierstrasz. Active objects in Hybrid. In *the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 243–253, 1987.

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

[22] H. Rajan. Capsule-oriented Programming. In *ICSE'15: The 37th International Conference on Software Engineering: NIER Track*, May 2015.

[23] 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.

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

[25] M. Rettig. Jetlang. http://code.google.com/p/jetlang/.

[26] C. Roig, A. Ripoll, and F. Guirado. A New Task Graph Model for Mapping Message Passing Applications. *IEEE Trans. Parallel Distrib. Syst. 2007*.

[27] C. Roig, A. Ripoll, M. A. Senar, F. Guirado, and E. Luque. Modelling Message-passing Programs for Static Mapping. In *EURO-PDP'00*.

[28] H. Sasaki, T. Tanimoto, K. Inoue, and H. Nakamura. Scalability-based Manycore Partitioning. In *PACT'12*.

[29] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends. In *DAC'13*.

[30] L. A. Smith, J. M. Bull, and J. Obdrzálek. A Parallel Java Grande Benchmark Suite. In *SC'01*.

[31] 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.

[32] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *ECOOP'08*.

[33] S. Tasharofi and R. Johnson. Actor Collection. http://actor-applications.cs.illinois.edu/.

[34] W. Thies and S. Amarasinghe. An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design. In *PACT'10*.

[35] A. Tousimojarad and W. Vanderbauwhede. An Efficient Thread Mapping Strategy for Multiprogramming on Manycore Processors. *Journal of Parallel Computing, 2014*.

[36] G. Upadhyaya and H. Rajan. An Automatic Actors to Threads Mapping Technique for JVM-based Actor Frameworks. In *AGERE'14*.

[37] J. Zhang. Characterizing the scalability of erlang vm on many-core processors. Master's thesis, KTH, 2011.