I dedicate this dissertation to my loving wife and supportive parents. Without you, I would not be here today.
ACKNOWLEDGMENTS

I would like to thank all of those wonderful people in my life who have supported me and pushed me to work harder. I will be forever grateful to my mom and dad for pushing me to get involved in school activities and not getting angry when my electronic experimenting resulted in the frying of our home computer. I am thankful for my teachers in grade school for realizing my passion for technology and encouraging me to excel. I would also like to thank my Ph.D. advisor, Dr. Greg Stitt, for all of his guidance and support throughout the last several years. Lastly, I would like to thank my wonderful wife, Mine, who always brings a smile to my face after a long day of work.

This work is financially supported by the National Science Foundation, grant CNS-0914474.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>4</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>7</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>8</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>10</td>
</tr>
<tr>
<td><strong>CHAPTER</strong></td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>11</td>
</tr>
<tr>
<td>2 RELATED RESEARCH</td>
<td>15</td>
</tr>
<tr>
<td>3 ELASTIC COMPUTING FRAMEWORK</td>
<td>19</td>
</tr>
<tr>
<td>3.1 Overview</td>
<td>19</td>
</tr>
<tr>
<td>3.2 Proposed Models of Usage</td>
<td>21</td>
</tr>
<tr>
<td>3.3 Elastic Functions</td>
<td>23</td>
</tr>
<tr>
<td>3.3.1 Implementations</td>
<td>25</td>
</tr>
<tr>
<td>3.3.2 Parallelizing Templates</td>
<td>27</td>
</tr>
<tr>
<td>3.3.3 Interface and Usage Assumptions</td>
<td>28</td>
</tr>
<tr>
<td>3.3.4 Adapter</td>
<td>29</td>
</tr>
<tr>
<td>3.4 Limitations</td>
<td>30</td>
</tr>
<tr>
<td>3.5 Summary</td>
<td>33</td>
</tr>
<tr>
<td>4 IMPLEMENTATION ABSTRACTION AND PERFORMANCE ANALYSIS</td>
<td>35</td>
</tr>
<tr>
<td>4.1 Overview</td>
<td>35</td>
</tr>
<tr>
<td>4.2 Adapters</td>
<td>36</td>
</tr>
<tr>
<td>4.2.1 Abstraction of Invocation Parameters</td>
<td>37</td>
</tr>
<tr>
<td>4.2.2 Creation of Work Metric Mappings</td>
<td>42</td>
</tr>
<tr>
<td>4.2.3 Design of Adapters</td>
<td>44</td>
</tr>
<tr>
<td>4.2.4 Limitations</td>
<td>46</td>
</tr>
<tr>
<td>4.3 Implementation Assessment with the IA Heuristic</td>
<td>47</td>
</tr>
<tr>
<td>4.3.1 Performance/Accuracy Tradeoff</td>
<td>51</td>
</tr>
<tr>
<td>4.3.2 Sample Collection Step</td>
<td>54</td>
</tr>
<tr>
<td>4.3.3 Segment Identification Step</td>
<td>58</td>
</tr>
<tr>
<td>4.3.4 Segment Insertion Step</td>
<td>64</td>
</tr>
<tr>
<td>4.3.5 Segment Commitment Step</td>
<td>66</td>
</tr>
<tr>
<td>4.4 Summary</td>
<td>68</td>
</tr>
<tr>
<td>5 ELASTIC FUNCTION PARALLELIZATION, OPTIMIZATION, AND EXECUTION</td>
<td>70</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-1</td>
<td>Invocation semantics, asymptotic analysis results, and resulting work metric mappings for possible convolution and matrix multiply adapters.</td>
<td>42</td>
</tr>
<tr>
<td>4-2</td>
<td>A description of the functions required for an adapter along with example functions for a sorting adapter.</td>
<td>44</td>
</tr>
<tr>
<td>4-3</td>
<td>A description of the IA Heuristic’s tunable parameters for controlling tradeoffs between performance and accuracy</td>
<td>51</td>
</tr>
<tr>
<td>5-1</td>
<td>The pointer movement rules used by segment adding.</td>
<td>83</td>
</tr>
<tr>
<td>6-1</td>
<td>Descriptions of the work metric mappings used for each function.</td>
<td>97</td>
</tr>
<tr>
<td>6-2</td>
<td>Work metric range and IPG creation time for the non-heterogeneous implementations.</td>
<td>98</td>
</tr>
<tr>
<td>6-3</td>
<td>Work metric range and IPG creation time for the heterogeneous implementations.</td>
<td>99</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>An overview of Elastic Computing, which is enabled by elastic functions that provide numerous implementations for performing a function.</td>
<td>19</td>
</tr>
<tr>
<td>3-2</td>
<td>The components of an elastic function for a sorting example.</td>
<td>24</td>
</tr>
<tr>
<td>4-1</td>
<td>High-level overview of the IA Heuristic.</td>
<td>35</td>
</tr>
<tr>
<td>4-2</td>
<td>Example of a sorting adapter mapping between invocation parameter-space and work metric-space.</td>
<td>38</td>
</tr>
<tr>
<td>4-3</td>
<td>An example lookup into an IPG for a sorting implementation.</td>
<td>48</td>
</tr>
<tr>
<td>4-4</td>
<td>An illustration of the four steps of the IA Heuristic.</td>
<td>49</td>
</tr>
<tr>
<td>4-5</td>
<td>An illustration of sample collection determining a work metric value to sample.</td>
<td>56</td>
</tr>
<tr>
<td>4-6</td>
<td>An illustration of the relationship between cells in the regression matrix and their corresponding samples.</td>
<td>59</td>
</tr>
<tr>
<td>4-7</td>
<td>An illustration of inserting a new sample into the regression matrix.</td>
<td>60</td>
</tr>
<tr>
<td>4-8</td>
<td>The region of cells to analyze as potential candidate segments.</td>
<td>62</td>
</tr>
<tr>
<td>4-9</td>
<td>An example of the three possible insertion locations for a segment.</td>
<td>64</td>
</tr>
<tr>
<td>4-10</td>
<td>An illustration of removing the top rows of the regression matrix during segment commitment.</td>
<td>66</td>
</tr>
<tr>
<td>5-1</td>
<td>The structure of a parallelizing template.</td>
<td>72</td>
</tr>
<tr>
<td>5-2</td>
<td>The high-level steps of the RACECAR heuristic.</td>
<td>73</td>
</tr>
<tr>
<td>5-3</td>
<td>Example of an IPG for a sorting implementation.</td>
<td>75</td>
</tr>
<tr>
<td>5-4</td>
<td>Example of creating a function performance graph from a set of implementation performance graphs.</td>
<td>77</td>
</tr>
<tr>
<td>5-5</td>
<td>Structure of a merge-sort parallelizing template.</td>
<td>79</td>
</tr>
<tr>
<td>5-6</td>
<td>Steps of creating a restricted parallelization graph.</td>
<td>82</td>
</tr>
<tr>
<td>5-7</td>
<td>Example of segment adding creating a sub-parallelization graph.</td>
<td>83</td>
</tr>
</tbody>
</table>
Example of a merge-sort parallelizing template performing a lookup in a parallelization graph.

An example of the function execution tool executing a sort for 10,000 elements on 4 threads/1 FPGA.

The estimation error of the IPG created by the IA Heuristic for 250 random invocations of each non-heterogeneous implementation.

The estimation error of the IPG created by the IA Heuristic for 250 random invocations of each heterogeneous implementation.

The speedup achieved by Elastic Computing for each elastic function.

The speedup achieved by Elastic Computing averaged over each system.
ELASTIC COMPUTING: A FRAMEWORK FOR EFFECTIVE
MULTI-CORE HETEROGENEOUS COMPUTING

By
John Robert Wernsing

December 2012

Chair: Greg Stitt
Major: Electrical and Computer Engineering

Due to power limitations and escalating cooling costs, high-performance computing systems can no longer rely on faster clock frequencies and more massive parallelism to meet increasing performance demands. As an alternative approach, high-performance systems are increasingly integrating multi-core processors and heterogeneous accelerators such as GPUs and FPGAs. However, usage of such multi-core heterogeneous systems has been limited largely to device experts due to significantly increased application design complexity. To enable more transparent usage of multi-core heterogeneous systems, we introduce Elastic Computing, which is an optimization framework where application designers invoke specialized elastic functions that contain a knowledge-base of implementation alternatives and parallelization strategies. For each elastic function, a collection of optimization tools analyze numerous possible implementations which enables dynamic and transparent optimization for different resources and run-time parameters. In this document, we present the enabling technologies of Elastic Computing, and evaluate those technologies on numerous systems, including the Novo-G FPGA supercomputer.
CHAPTER 1
INTRODUCTION

For the past several decades, the high-performance computing (HPC) community has relied on rapidly increasing clock frequencies and increasing parallelism at a massive scale to meet the escalating performance demands of applications. However, such an approach has become less feasible due to several limitations. Clock frequencies have slowed due to power and cooling limits of integrated circuits, and it is becoming economically infeasible to increase system sizes due to energy and cooling costs [16], which are becoming the dominant factor in the total cost of ownership [18].

To overcome these limitations, HPC systems have started on a trend towards increased heterogeneity, with systems increasingly integrating specialized microprocessor cores [24], graphics processing units (GPUs) [34][58], and field-programmable gate arrays (FPGAs) [7][18]. Such multi-core heterogeneous systems tend to provide improved energy efficiency compared to general-purpose microprocessors [54][62] by using specialization to significantly reduce power requirements while also improving performance [11][22]. As a motivating example, the Novo-G supercomputer [18], which uses 192 FPGAs in 24 nodes, has achieved speedups of more than 100,000x compared to a 2.4 GHz Opteron system for computational biology applications [18]. Such speedup provides performances similar to Roadrunner and Jaguar – two of the top supercomputers – even when assuming perfect linear scaling of performance for additional cores on those machines [57]. Furthermore, traditional supercomputers typically require an average of 1.32 megawatts of power [56], whereas Novo-G consumes a maximum of 8 kilowatts.
Although multi-core heterogeneous systems provide significant advantages compared to traditional HPC systems, effective usage of such systems is currently limited by significantly increased application design complexity that results in unacceptably low productivity [41]. While parallel programming has received much recent attention, the inclusion of heterogeneous resources adds additional complexity that limits usage to device experts. For example, with an FPGA system, application designers often must be experts in digital design, hardware description languages, and synthesis tools. GPU systems, despite commonly being programmed with high-level languages, share similar challenges due to architecture-specific considerations that have significant effects on performance [43].

Numerous design automation and computer-aided design studies have aimed to reduce design complexity for multi-core heterogeneous systems by hiding low-level details using advanced compilation [6][19][26] and high-level synthesis [23][28][37][52] combined with new, specialized high-level languages [28][47]. Although these previous approaches have had some impact on productivity, a fundamental limitation of previous work is the attempt to transform and optimize the single implementation specified by the application code. Much prior work [20][55] has shown that different implementations of the same application often have widely varying performances on different architectures. For example, a designer implementing a sorting function may use a merge-sort or bitonic-sort algorithm for an FPGA but a quick-sort algorithm for a microprocessor. Furthermore, this algorithmic problem extends beyond devices; different algorithms operate more efficiently for different input parameters [40], different amounts of resources [17], and potentially any other run-time parameter. Although existing tools
can perform transformations to optimize an implementation, *those transformations cannot convert between algorithms* (e.g., quick-sort into merge-sort), which is often required for efficiency on a particular device or for different numbers of devices. Thus, even with improved compilers, synthesis tools, and languages, efficient application design for multi-core heterogeneous systems will still require significant designer effort, limiting usage to device experts.

To address these limitations, we propose a complementary approach, referred to as *Elastic Computing*, which enables transparent and portable application design for multi-core heterogeneous systems while also enabling adaptation to different run-time conditions. Elastic Computing is an optimization framework that combines standard application code – potentially written in any language – with specialized elastic functions and corresponding optimization tools. Much of the novelty of Elastic Computing is enabled by elastic functions, which provide a knowledge-base of implementation alternatives and parallelization strategies for a given function. When an application calls an elastic function, the Elastic Computing tools analyze available devices and resources (e.g., CPU cores, GPUs, FPGAs) and current run-time parameters (e.g., input size), and then transparently select from numerous pre-analyzed implementation possibilities. For the sorting example, an application designer using Elastic Computing would simply call an elastic sorting function without specifying how that sort is implemented, which the Elastic Computing tools would parallelize across available resources while selecting appropriate implementations for each resource. Thus, without any effort or knowledge of the architecture, the application designer in this example is able to execute a sorting implementation that effectively takes advantage of up to all the heterogeneous
resources on a system. While previous work has shown such optimization for specific systems, applications, and languages, to our knowledge, Elastic Computing is the first technique that potentially enables transparent optimization of arbitrary functions for any run-time condition on any multi-core heterogeneous system.

The organization of the remainder of this document is as follows. Related prior research is discussed in Chapter 2. Chapter 3 provides a high-level overview of the Elastic Computing Framework, including the steps of the framework and the components of elastic functions. Next, Chapter 4 discusses the details of transparently analyzing the performance of implementations, which is the first of the two main optimization steps of the framework. Chapter 5 then continues with a discussion on parallelizing, optimizing, and executing elastic functions, which internally relies on the performance estimation techniques presented in Chapter 4. Chapter 6 then presents experimental results of the effectiveness of Elastic Computing. Lastly, Chapter 7 presents the conclusions of this research.
CHAPTER 2
RELATED RESEARCH

Codesign-extended applications [20] share similarities with Elastic Computing by allowing designers to specify multiple implementations of a function, which enables a compiler to explore multiple possibilities for hardware and software implementations. Although this approach achieves improvements in portability and efficiency, application designers have to manually specify the behavior of multiple implementations and when the compiler should use those implementations, resulting in decreased productivity.

With Elastic Computing, for cases where an appropriate elastic function is provided, application designers do not specify any implementation details and instead simply call the elastic function, with efficient implementations of that function determined automatically by the Elastic Computing Framework. In addition, Elastic Computing can combine and parallelize existing implementations to create new implementation possibilities based on run-time conditions.

Numerous compiler studies have focused on automatic parallelizing transformations [13][19][27] and adaptive optimization techniques [6][31] to optimize applications for different multi-core architectures. For FPGAs, high-level synthesis tools [23][28][37] have focused on translating and optimizing high-level code into custom parallel circuit implementations. For GPUs, frameworks such as CUDA [43], Brook [5], and OpenCL [29] provide an extended version of the C language that allows code to be compiled onto both CPUs and GPUs. While these tools have simplified the portability of implementations between devices, their efficiency is still fundamentally limited by the single algorithm described in the original specification. Elastic Computing is complementary to these tools, enabling multiple implementations written in any
language and using any algorithm to transparently work together to improve the efficiency of applications.

Performance prediction, analysis, and simulation techniques are widely studied topics that share a subset of Elastic Computing’s challenges. Existing performance prediction techniques are often used to evaluate the amenability of a particular architecture for certain applications [2][25][48], to assist design space exploration [35] and verification [32], to help identify performance bottlenecks [42][53], among others. Although the majority of previous work focuses on microprocessors, other approaches focus on performance prediction for FPGAs using analytical [25] and simulation [51] methods. Some previous work also allows for full-system simulation and prototyping [21]. Ptolemy [14] is a tool that supports multiple-modes of computation and complex hierarchical designs for prototyping and simulation. Although existing performance prediction and simulation techniques are related to Elastic Computing in that they allow for the assessment of a design for an application, those techniques do not automatically consider implementation or execution alternatives and therefore optimizing the application still requires designer effort. Much previous work has focused on design-space exploration to automate this optimization [15][39][46], but those approaches do not consider alternative algorithms, work partitionings, and device partitionings for all possible devices, input parameters, and usage assumptions. Elastic Computing does not require any manual optimization and automatically explores different ways for executing an application before selecting the best performing.

Previous work on adaptable software and computation partitioning also shares similarities with Elastic Computing. FFTW (Fastest Fourier Transform in the West) [17]
is an adaptive implementation of FFT that tunes an implementation by composing small blocks of functionality, called codelets, in different ways based on the particular architecture. OSKI (Optimized Sparse Kernel Interface) [59] is a similar library of automatically-tuned sparse matrix kernels. ATLAS [61] is a software package of linear algebra kernels that are capable of automatically tuning themselves to different architectures. SPIRAL [49] is a similar framework but explores algorithmic and implementation choices to optimize DSP transforms. Such approaches essentially perform a limited form of Elastic Computing specific to microprocessor architectures and using application-specific optimization strategies. PetaBricks [1] consists of a language and compiler that enables algorithmic choice, but restricts parallelizing decisions to static choices. Qilin [33] can dynamically determine an effective partitioning of work across heterogeneous resources, but targets data-parallel operations. MapReduce is a programming model that allows for the automatic partitioning of data-intensive computation across nodes in a cluster [10] or resources in a system [50], however it is only useable by computation that adheres to the map-reduce structure. Elastic Computing aims to provide a general framework that enables any functionality to be optimized for any architecture, as well as supporting the dynamic parallelization of work across different resources. Also, whereas previous work has focused primarily on microprocessor architectures, Elastic Computing can potentially be used with any multi-core heterogeneous system and can also adapt to run-time changes.

Resource-constrained scheduling is similar in purpose to Elastic Computing. Numerous exact and heuristic algorithms [9] exist for scheduling a task graph onto a fixed set of resources. Some previous works also tailor these algorithms for specific
types of devices. SPARCS [45] schedules task graphs temporally and spatially onto a set of FPGAs. While the purpose is similar, Elastic Computing supports elastic functions with unpredictable execution and thereby is not limited to the static task graph abstractions required by scheduling algorithms. Instead, elastic functions have the flexibility to provide numerous implementation and parallelization alternatives from which the Elastic Computing Framework determines the best performing based on the actual run-time conditions.
In this chapter, we describe the main components of Elastic Computing. Section 3.1 presents an overview of the Elastic Computing Framework and its high-level components. Section 3.2 proposes several models of usage for Elastic Computing. Section 3.3 describes the components of elastic functions. Lastly, Section 3.4 presents a summary of the main limitations of Elastic Computing.

### 3.1 Overview

Figure 3-1. An overview of Elastic Computing, which is enabled by elastic functions that provide numerous implementations for performing a function. At (a) install time, implementation assessment and optimization planning create estimators of performance to compare alternate implementations and optimizations. At (b) run time, an application calls an elastic function, which starts the function execution tool to determine the best way to execute the elastic function by referring to information collected during install time.

Elastic Computing, overviewed in Figure 3-1, is an optimization framework that combines specialized elastic functions with tools for implementation assessment, optimization planning, and function execution. To invoke Elastic Computing, application designers include calls to elastic functions from normal application code. Elastic functions are the fundamental enabling technology of Elastic Computing, which consist
of a knowledge-base of alternative implementations of the corresponding functionality (e.g., quick-sort on CPU, merge-sort on GPU, bitonic-sort on FPGA) in addition to parallelizing templates, which specify strategies for how to parallelize computation across multiple resources (e.g., partition a large sort into multiple sub sorts and execute each sub-sort in parallel). By integrating this additional knowledge of a function, elastic functions enable the Elastic Computing tools to explore numerous implementation possibilities that are not possible from a single high-level specification (i.e., normal application code), while also enabling automatic adaptation for different run-time conditions such as available resources and input parameters.

Although Elastic Computing could potentially determine the most effective implementation of each elastic function call completely at run-time, such an approach would likely have prohibitive overhead. To reduce this overhead, Elastic Computing instead performs two analysis steps at installation-time – implementation assessment (Section 4.3) and optimization planning (Section 5.2) – to pre-determine efficient implementations for different execution situations.

As shown in Figure 3-1 for the sorting example, implementation assessment builds a data structure that estimates the execution time of each implementation on each device for different input parameters. Implementation assessment builds the data structure by repeatedly executing each implementation, measuring the execution time, and statistically analyzing the results.

After implementation assessment, optimization planning then creates a new data structure which uses the results from implementation assessment to determine which implementations to use for different execution situations as well as how to parallelize
computation across multiple resources. Optimization planning creates the data structure by determining execution decisions that minimize the estimated execution time as predicted by the results of implementation assessment.

At run-time, when an application invokes an elastic function, the function execution tool (Section 5.4) selects and executes efficient implementations of the elastic function for the current execution situation. Because optimization planning already determined efficient execution decisions at installation-time, the function execution tool simply refers to the optimization planning results when determining which implementations to select. For elastic functions which provide parallelizing templates, the function execution tool may also parallelize computation across multiple resources. As illustrated by the figure, the application simply calls an elastic sorting function, which the function execution tool then transparently partitions into a quick-sort algorithm running on the microprocessor and a bitonic-sort algorithm running on the FPGA, with additional post-processing to combine the results from each device.

In summary, Elastic Computing enables portable application design for multi-core heterogeneous systems by transparently optimizing the execution of elastic functions to the input parameters of a function call and the devices on a system.

### 3.2 Proposed Models of Usage

To take advantage of Elastic Computing, applications must call elastic functions, which raise the question of how those elastic functions are created or provided. In this section, we summarize two envisioned usage models. In practice, we expect a combination of these two models to be used.

The main target for Elastic Computing is mainstream, non-device-expert application designers who are solely concerned with specifying functionality and often
lack the expertise required to design efficient functions for multi-core heterogeneous systems. Motivating examples include domain scientists, such as computational biologists/chemists, who commonly write applications in C/Fortran and MPI, and would like to use multi-core heterogeneous systems without becoming experts in FPGA’s, GPU’s, etc. [38]. Because creating implementations for an elastic function often requires the same skills as creating device-specific implementations, such designers will not be able to create their own elastic functions. To support these designers, we envision a library usage model, where ideally, application designers call elastic functions in the same way as existing, widely-used function libraries. Elastic function libraries could potentially be created for different application domains, where implementations of each elastic function are provided by several potential sources. Device vendors are one likely source of implementations, because by enabling transparent usage of their corresponding devices via elastic functions, those devices could potentially become useable by new markets. In fact, many hardware vendors, such as Intel for their microprocessors [29], Xilinx for their FPGAs [63], and Nvidia for their GPUs [44], already provide optimized code for common functions executing on their devices. By instead providing optimized code in the form of elastic function implementations, or by having third parties port this code into elastic functions, their device could work in conjunction with all the other heterogeneous resources on a system to further provide speedup. Third-party library designers (e.g., Rapidmind [36]), who already target specialized devices, could also provide implementations of elastic functions for different devices. Finally, open-source projects (e.g., FFTW [17], ATLAS [61], OSKI [59]) could potentially establish a standardized elastic function library that could be extended with
implementations from all participants. In this situation, experts from different domains could provide implementations optimized for different situations and devices. With the ability of Elastic Computing to automatically incorporate implementations, mainstream application designers could transparently exploit an implementation specialized for any given situation without any knowledge of the actual implementation or situation.

Of course, there may be applications where designers must create new implementations either to target new hardware or provide new functionality. Although this usage model does often require device-specific knowledge, Elastic Computing aids the development of new implementations by allowing an implementation to be defined using existing elastic functions and by automatically determining efficient execution and parallelizing decisions. For example, if a new elastic function requires sorting functionality (e.g., convex hull), an implementation of that function could call an elastic sort function, for which Elastic Computing already determines how to execute efficiently on a system. In addition, the function execution tool provides communication and resource management features, which simplify the creation of multi-core and/or heterogeneous implementations.

3.3 Elastic Functions

Elastic functions define multiple implementations of a given functionality, which enables the Elastic Computing tools to perform a detailed exploration of potential optimizations for different systems. There is no restriction on what kind of functionality elastic functions can describe, but to maximize the performance potential of Elastic Computing, elastic functions should ideally be specified using alternate algorithms that are appropriate for different devices and/or input parameters. Additionally, elastic
functions should provide parallelizing templates which specify how to parallelize computation across multiple resources.

Elastic functions can be defined at any granularity, ranging from simple operations, such as sorting an array or performing inner-product on two vectors, to more complicated operations, such as frequency-domain convolution or matrix multiply. Elastic functions may even internally call nested elastic functions, which enable a huge potential exploration space for optimization that can provide increased performance benefits and applicability on a wider range of systems.

As illustrated in Figure 3-2, each elastic function consists of a set of implementations, a set of parallelizing templates, an interface, a set of usage assumptions, and an adapter. As discussed in the following sections, the implementations provide the alternate ways Elastic Computing may perform the functionality of the elastic function. The parallelizing templates specify how Elastic
Computing may parallelize the computation of the function across multiple resources.
The *interface* specifies how an application invokes the elastic function, which the
application developer uses in conjunction with the *usage assumptions* to specify greater
detail about the invocation. Lastly, the *adapter* abstracts the details of the
implementations to enable the Elastic Computing tools to support analyzing and
optimizing nearly any kind of elastic function and implementation.

### 3.3.1 Implementations

Each elastic function may specify several alternate implementations to perform the
functionality of an elastic function, as illustrated in Figure 3-2. Each implementation is
functionally identical (i.e., has the same inputs and outputs), but may rely on different
algorithms and/or use different amounts and types of hardware resources. For example,
a sorting elastic function may have one implementation that uses an insertion-sort
algorithm executing on a single CPU thread, while a second implementation uses a
multi-threaded version of a merge-sort algorithm, while a third implementation uses a
bitonic-sort algorithm executing on an FPGA. At run-time, the function execution tool will
select the implementation that it estimates will provide the best performance for a given
invocation or, potentially, have multiple implementations execute in parallel.

There are few restrictions on the code used to define the implementations. The
implementations can be written in any language that can be compiled and linked with
the framework (e.g., C/C++, assembly, FORTRAN). The implementation code can
access files, sockets, and interface with drivers. The only restriction on the code is that
the implementations must be thread-safe, as the function execution tool may spawn
multiple instances of the same implementation across different threads.
In addition to the code that defines functionality, the implementations also specify what resources the code requires for execution. When the function execution tool executes an implementation, the tool verifies that the required resources are available and prevents multiple implementations from simultaneously using the same resources. An implementation may also specify that it uses a variable number of resources. For example, many multi-threaded implementations evenly distribute work across any number of threads, allowing the same implementation code to be equally valid for any number of threads allocated to it. In these cases, the implementations need only specify the minimum number of resources required.

Elastic Computing, which is designed to be independent of specific heterogeneous resources, only directly controls the thread execution of software implementations, but the implementation code may itself configure and make use of any heterogeneous resources on a system (e.g., through a corresponding driver interface). Therefore, thread implementations may reference supplementary device-specific code such as compiled VHDL or CUDA code. The only requirement is that the implementation must specify the heterogeneous devices it requires so that the function execution tool will prevent simultaneously executing implementations from using the same resource. It is a future work task to enable native support for heterogeneous resources, likely by automating the creation of the corresponding thread function to configure and use the heterogeneous resource on behalf of the heterogeneous implementation.

Elastic Computing fully supports multi-threaded implementations and provides facilities for multi-threaded cooperation, with functionality similar to MPI (Message Passing Interface). When the function execution tool selects multiple threads to execute
a multi-threaded implementation, the function execution tool automatically spawns instances of the implementation on each of the threads. Within each thread, the implementation may access the number of fellow thread instances as well as the current thread’s relative index (i.e., an MPI rank). The function execution tool additionally provides functionality to communicate and synchronize individually or collectively with the fellow instances. One benefit of multi-threaded implementations is that the implementation instances execute in the same address space, allowing for the efficient passing of data between threads using pointers (as opposed to having to transfer entire data structures). Elastic Computing also supports multi-threaded heterogeneous implementations, with the function execution tool similarly only spawning instances of the implementation on the threads and relying internally on the implementation code to make use of the heterogeneous resources.

Implementations may also make nested elastic function calls to the same or another elastic function. Identical to how traditional complex functions are often defined using numerous simpler functions, implementations of complex elastic functions can be defined using more basic elastic functions. For example, one implementation of convex hull internally relies on a sorting step. As opposed to statically defining the code for the sort (or even calling a sort function from a standard library), an implementation can internally invoke a sort elastic function, enabling the implementation to take advantage of the optimizations for the sort.

3.3.2 Parallelizing Templates

Parallelizing templates specify how to parallelize the computation of an elastic function across two distinct subsets of resources. While each parallelizing template only partitions computation into two, Elastic Computing may nest parallelizing templates
arbitrarily deep to create as many parallel executions as would yield a performance benefit, utilizing up to all of the resources on a multi-core heterogeneous system.

The basic structure of a parallelizing template mimics the design of a divide-and-conquer algorithm and essentially transforms one large execution of a function into two smaller, independent executions of the same function. For any function which supports a divide-and-conquer algorithm (e.g., merge-sort for a sorting elastic function), a developer may create a corresponding parallelizing template. The structure of the parallelizing template is to first partition the input parameters into two smaller sets of input parameters, invoke two nested instances of the elastic function to calculate the output for the two smaller sets of input parameters, and then merge the outputs of the two nested elastic functions when complete.

Elastic computing heavily optimizes the execution of parallelizing templates by referring to decisions made during the optimization planning step. The decisions specify both how to partition the computation and which implementations/resources to utilize for each of the two nested elastic function calls. More information on the structure of parallelizing templates and how Elastic Computing determines the optimizations are described in Chapter 5.

3.3.3 Interface and Usage Assumptions

The elastic function interface is the mechanism that applications use to invoke the elastic function. From an application designer’s point of view, the interface is identical to a normal function call, and is defined by a function prototype that contains the parameters needed by the function. The interface itself does not directly perform any of the functionality of the elastic function, but instead simply launches the function execution tool to execute the elastic function, as described in Section 5.4. Once the
elastic function is complete, control returns to the interface, which populates any output parameters before returning control to the application.

The interface also specifies a list of usage assumptions to allow the application designer to provide extra information about the characteristics of the input parameters. For many elastic functions, some specific instances of input parameters form corner cases that require different optimization decisions. For example, a sorting elastic function may provide usage assumptions allowing for the specification of data characteristics (e.g., randomly distributed, mostly sorted, or reverse sorted). This extra information helps the Elastic Computing tools make better optimization decisions, perhaps choosing an algorithm such as quick-sort for randomly distributed input data as opposed to insertion-sort for mostly sorted input data. Internally, the interface treats each usage assumption as independent cases, allowing for completely different optimization decisions for different usage assumptions. The application specifies the usage assumption by passing a parameter in the elastic function call.

3.3.4 Adapter

The adapter is responsible for abstracting the details of the implementations in order to allow for Elastic Computing to support analyzing and optimizing nearly any kind of elastic function. As Elastic Computing was not designed for any specific type of computation, a wide variety of elastic functions may be created. Each elastic function may require different types of input/output parameters and have drastically different invocation and execution characteristics from any other. For this reason, if adapters were not used, multiple versions of the optimization tools would need to be created for each type of elastic function. As a result, adapters provide an abstraction layer that
isolate implementation-specific details and provide Elastic Computing with an implementation-independent interface.

The main purpose of the adapter is to abstract the details of the input parameters for an implementation by mapping the parameters to an abstract representation called the *work metric*. The mapping between input parameters and work metric is specific to an elastic function, requiring a separate adapter for each elastic function. Despite this additional effort, the adapters themselves are very simple to design and create, as described in detail in Section 4.2.

As the adapters map all of the implementations to a similar work metric interface, the implementation assessment and optimization planning tools need only support the work metric interface to allow for the analysis and optimization of nearly any kind of elastic function. Section 4.3 describes how implementation assessment analyzes implementations using the work metric interface. Lastly, Section 5.3 describes how optimization planning determines efficient execution decisions using the work metric interface.

### 3.4 Limitations

The main limitation of Elastic Computing is that the improvement in performance and design productivity depends on the percentage of code that can be defined using elastic functions. Ideally, an elastic function library combined with vendor-provided implementations could provide most designers with the majority of functionality they require, but reaching this level of maturity will take time. Until then, developers would need to manually create elastic functions for which Elastic Computing still provides numerous benefits in terms of a runtime environment, automatic work parallelization, and a framework that supports code reuse. Any elastic function would only need to be
created once, and then reused by other developers for different applications or executing on different systems. Additionally, Elastic Computing allows developers to incrementally improve elastic functions by adding new or improved implementations that support new system resources or execute more efficiently. Any application which uses an elastic function would automatically incorporate the latest improvements to the elastic function, as described in Section 5.3.

Another limitation of Elastic Computing is that the efficiency and availability of implementations limit the speedup of the elastic functions. The Elastic Computing Framework has no internal understanding of how to perform any of the computation for a function, and therefore must rely solely on the provided implementations to actually perform the function’s computation. As a result, the selection of implementations fundamentally limits the speedup of the elastic function, with the efficiency of the implementations determining the function’s efficiency and the availability of implementations to execute on a resource limiting what resources the function may utilize. Despite this limitation, the framework does determine how to best utilize the available implementations to maximize performance. As described in Section 5.3, Elastic Computing will only select the implementations that it estimates to be the most efficient and will apportion work appropriately between the implementations to better utilize higher performing implementations. Note that even an un-optimized implementation will still provide overall speedup if it executes in parallel with other implementations. For example, even if an un-optimized CUDA implementation provides only a 1x speedup when compared to a CPU implementation, having the elastic function utilize both implementations executing in parallel may achieve a 2x speedup over the
CPU by itself. Also note that this limitation is not specific to Elastic Computing as any application is fundamentally limited by the efficiency of the code performing the computation and the availability of code to execute on the different system resources. However, Elastic Computing improves on the situation by automatically deciding how to best utilize the provided implementation to improve function performance on the system.

Lastly, Elastic Computing requires the implementations to be of the form of a function definition, which has the side effect that Elastic Computing must pass the input/output parameters into the function using normal function call semantics. This may be a limitation for certain types of computation that may be structured more efficiently otherwise. While implementations must currently adhere to this function call structure, Elastic Computing does not need to duplicate the parameters for each function call as it may instead pass pointers of data structures common to multiple threads of execution. For resources that may not directly access the system memory (e.g., FPGAs, GPUs), the input/output parameters will need to be copied to/from the resource during the course of the execution leading to some additional overhead. However, this requirement is not unique to Elastic Computing and would be required of any computation that uses heterogeneous resources in general. Some improvement in performance may be possible for the case when multiple implementations execute on the same heterogeneous resource back-to-back if intermediate results could be saved on the resource instead of copied off and then copied back on. This approach would require knowing a priori the future executions on a particular resource and identifying the common data flows to eliminate redundant transfer overhead, which is a potential area of future work.
3.5 Summary

In this chapter, we provided an overview of Elastic Computing and elastic functions. Elastic Computing is an optimization framework that aims to reduce the difficulty of developing efficient applications on multi-core heterogeneous systems. Elastic Computing consists of a library of elastic functions, each of which performs the computation of a single function (e.g., sort, matrix multiply, FFT) with extra information that Elastic Computing uses to optimize that function onto a multi-core heterogeneous system. Application developers treat Elastic Computing as a high-performance auto-tuning library of functions and simply invoke elastic functions from their code when they require the corresponding computation.

Each elastic function contains a knowledge-base of implementation alternatives and parallelization strategies for performing a function. The different implementation alternatives, each of which may execute using a different algorithm or on different system resources, provide Elastic Computing with execution alternatives from which it will select the best performing for a specific execution situation. The parallelization strategies, specified using parallelizing templates, provide Elastic Computing with knowledge of how to partition the computation across multiple parallel resources.

Elastic Computing determines how to most efficiently execute an elastic function at installation-time by performing the steps of implementation assessment and optimization planning. Implementation assessment creates a data structure that estimates the execution time of an implementation from its input parameters. Optimization planning then collectively analyzes all of the implementation assessment results to predetermine execution decisions that minimize the overall estimated
execution time. Both steps save their results as data structures for efficient reference at run-time when an application invokes an elastic function.

Chapter 4 continues on by describing the abstraction of implementations and the details of implementation assessment. Following which, Chapter 5 describes parallelizing templates and the details of optimization planning and elastic function execution. Lastly, Chapter 6 presents speedup results achieved by Elastic Computing for several elastic functions.
CHAPTER 4
IMPLEMENTATION ABSTRACTION AND PERFORMANCE ANALYSIS

In this chapter, we describe how Elastic Computing assesses the performance of implementations. Section 4.1 provides an overview of implementation abstraction and performance analysis. Section 4.2 describes how adapters abstract implementation-specific details so that implementation assessment can analyze many different types of implementations without modification. Section 4.3 then describes the IA Heuristic, which is the algorithm Elastic Computing uses to perform implementation assessment. Measurements of the accuracy and efficiency of the IA Heuristic are presented in Chapter 6.

4.1 Overview

Assessing the execution time performance of an implementation, also called implementation assessment, is the first of two installation-time optimization steps performed by Elastic Computing. During implementation assessment, Elastic Computing analyzes the performances of the different implementations of an elastic function to determine which implementation performs best in different execution situations. The result of this analysis is then used to make efficient execution decisions for the elastic function during the optimization planning step, which is discussed in Chapter 5.

![IA Heuristic Diagram](image)

Figure 4-1. High-level overview of the IA Heuristic.

The Implementation Assessment (IA) Heuristic is an efficient and flexible heuristic that analyzes an implementation and creates a data structure, referred to as an
**implementation performance graph (IPG)**, to estimate the execution time of the implementation on a particular device from the implementation’s invocation parameters. As illustrated in Figure 4-1, the IA Heuristic analyzes the implementation through a provided implementation-specific abstraction layer, called an *adapter*, that abstracts many of the details of the implementation and allows the heuristic to transparently support many different types of implementations. The details of how to create an adapter are discussed in Section 4.2.

The IA Heuristic, discussed in Section 4.3, analyzes the implementation, through the adapter, by statistically analyzing the execution time required for the implementation to complete for a variety of different input parameters, in a process we refer to as *sampling*. As the heuristic merely invokes the implementation (as opposed to performing an internal analysis of the implementation’s code), the heuristic treats the implementation as a black-box, which allows the heuristic to support nearly any kind of implementation, written in any programming language, and executing on any type of device. During the analysis, the heuristic adapts the sampling process in an attempt to minimize the total number of samples without sacrificing the accuracy of the resulting IPG. When complete, optimization and other analysis tools may then refer to the resulting IPG to estimate the performance of the implementations for different input parameters to inform their optimization processes.

### 4.2 Adapters

The IA Heuristic relies on abstraction to be widely applicable for different types of implementations and to isolate implementation-specific details. One of the main goals of the IA Heuristic is to support any type of implementation; however, the challenge is that different implementations may have significantly different invocation semantics.
Invocation details such as the structure of invocation parameters, the procedure to allocate and initialize parameters with valid values before an invocation, and any associated de-allocation of parameters following an invocation are all specific to an implementation and may vary widely between different types of implementations. For example, if the IA Heuristic was analyzing a sorting implementation, the heuristic would need to allocate and populate an input array for which the implementation to sort. However, if instead the function was matrix multiply, then the heuristic would need to instead create two matrices (of compatible dimensions) for the implementation to multiply. As such, if abstraction were not used, supporting different types of implementations would require creating separate versions of the heuristic, significantly hindering usability and practicality.

As a result, the IA Heuristic instead separates out the implementation-specific details by means of an abstraction layer, called an adapter, which transforms the interface of an implementation into a consistent generic interface. The adapter is specific to an implementation, requiring a developer knowledgeable of the implementation to create the adapter. However, the adapter needs only to be created once per implementation, likely by the same developer who initially creates the implementation. With the adapter, the IA Heuristic needs only to support the adapter’s generic interface to support analyzing any implementation.

4.2.1 Abstraction of Invocation Parameters
The most important feature of an adapter is to abstract away the specifics of the invocation parameters for an implementation. Different implementations may require significantly different invocation parameters, such as a sorting implementation requiring a single input array to sort as opposed to a matrix multiply implementation requiring two
matrices to multiply. Finding a consistent representation for all possible invocation parameters requires simplifying the parameters, without sacrificing too much accuracy when estimating the performance of an implementation.

Figure 4-2. Example of a sorting adapter mapping between invocation parameter-space and work metric-space.

Adapters use the abstraction of mapping an implementation's invocation parameters to a numeric value, called the work metric, which approximately measures the amount of computational work required by the implementation to process those parameters. The definition of computational work is very flexible and is chosen by the developer of an adapter to make the mapping from invocation parameters to work metric efficient to calculate. The measure of computational work does not need to be proportional to the execution time but should generally be a monotonically non-decreasing function of the execution time such that invocation parameters with a higher measure of computational work require a longer execution time. For the sorting example, a possible estimate for the amount of computational work is the number of elements to sort. Thereby, a potential work metric mapping is to set the work metric equal to the number of elements in the input array, as illustrated in Figure 4-2. The creation of a work metric mapping for the matrix multiply implementation is not as
straight-forward due to its input parameters requiring two matrices, however in this case (and in many cases) an effective work metric mapping can still be determined by analyzing how the execution time relates to the invocation parameters, as discussed in Section 4.2.2.

There are only two guidelines for effective work metric mappings. First, invocation parameters that map to the same work metric value, called *work metric groups*, should also require approximately the same execution time, as illustrated in Figure 4-2. Second, invocation parameters with larger work metric values should generally require longer execution times. For the sorting example, using the number of elements to sort as the work metric generally adheres to both of these guidelines, as the execution time required to sort an input array is typically not significantly dependent on the values contained within the array (assuming the input is randomly-distributed) and sorting more elements typically takes longer. Note that adapters are not required to perfectly adhere to these guidelines, but the resulting accuracy of the IA Heuristic is dependent upon the quality of the work metric mapping. A widely applicable technique to create effective work metric mappings is discussed in Section 4.2.2.

The mapping from invocation parameters to work metrics provides numerous practical and performance benefits to the IA Heuristic. First, the mapping reduces the potentially complex combinations of valid input parameters, which we refer to as the *invocation parameter-space*, of an implementation to a single numeric quantity (i.e., the work metric), which simplifies the design and operation of the IA Heuristic. Second, any implementation that has an adapter will support a consistent work metric interface, allowing the heuristic to operate identically regardless of the implementation. Lastly, the
mapping effectively reduces the dimensionality of the invocation parameter-space to a single dimension, which greatly reduces the amount of processing required by the IA Heuristic as it does not need to explore the entire invocation parameter-space. Note that although this reduction in dimensionality does limit the effectiveness of the heuristic for some functions, as described in Section 4.2.4, the goal of the mapping is to only coalesce (i.e., map to the same work metric value) invocation parameters that require approximately the same execution time, thereby eliminating only the insignificant dimensions for purposes of the IA Heuristic. In practice, no work metric mapping is perfect and likely some variance in execution time will exist for coalesced invocation parameters (i.e., work metric groups). However, for many implementations, the variance is not significant percentage-wise, as illustrated in the results in Section 6.2. Additionally, Section 4.2.4, also presents some mitigation techniques to reduce the variance of execution time in work metric groups.

In addition to the forward mapping from invocation parameters to work metric, the IA Heuristic also requires the adapter to specify the inverse mapping from work metric to invocation parameters. As the forward mapping is typically many-to-one, the inverse mapping requires the selection of a representative instance out of the invocation parameters within a work metric group. The IA Heuristic uses the inverse mapping in a process called sample collection (discussed in Section 4.3.2) to establish the execution time of invocation parameters with a specified work metric value. As a result, a representative instance should correspond to invocation parameters that have an execution time typical (i.e., not a corner-case) for that work metric value. For cases where the work metric mapping perfectly coalesces only invocation parameters with
identical execution times, any instance of invocation parameters within that work metric group would suffice. However, for the more common case of the coalesced invocation parameters having a distribution of execution times, the inverse mapping should ideally return an instance of invocation parameters with an execution time close to the average execution time for the group. For the sorting example, a representative instance of the invocation parameters for a specified work metric value might be to create a randomly initialized input array with length equal to that work metric value (assuming that an input array with randomly distributed values is representative of the typical invocation parameters).

Most forward and inverse mappings inherently require knowledge of what are typical invocation parameters for a function in order for the adapter to remain accurate across the expected input parameter space. For this reason, Elastic Functions provide usage assumptions, described in Section 3.3.3, to allow the user to specify more information about the type of invocation. For the sorting example, it was implied that a randomly distributed input array was the typical case. However, for some applications the typical case may instead be a mostly sorted input array, resulting in significantly different execution times depending on the implementation. In this case, the Elastic Function may provide a different usage assumption for the randomly distributed and mostly sorted cases. Section 4.2.4 provides more information about the limitations of adapters and mitigation techniques, including incorporating usage assumptions.
4.2.2 Creation of Work Metric Mappings

Table 4-1. Invocation semantics, asymptotic analysis results, and resulting work metric mappings for possible convolution and matrix multiply adapters.

<table>
<thead>
<tr>
<th></th>
<th>Convolution Adapter (for time-domain algorithm)</th>
<th>Matrix Multiply Adapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invocation</td>
<td>Convolve(Array a, Array b)</td>
<td>MultiplyMatrices(Matrix a, Matrix b)</td>
</tr>
<tr>
<td>Semantics:</td>
<td>(convolve array a with array b)</td>
<td>(multiply matrix a with matrix b)</td>
</tr>
<tr>
<td>Asymptotic</td>
<td>time = $\Theta(</td>
<td>a</td>
</tr>
<tr>
<td>Analysis:</td>
<td>(where $</td>
<td>a</td>
</tr>
<tr>
<td>Work Metric</td>
<td>work metric = $</td>
<td>a</td>
</tr>
<tr>
<td>Mapping:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

One technique a developer of an adapter may use to create an effective work metric mapping is to base the mapping on the results of an asymptotic analysis of the implementation. An asymptotic analysis reveals how the invocation parameters of an implementation influence the implementation’s execution time. For example (and as described in Table 4-1), the asymptotic analysis for a convolution implementation, based on the discrete time-domain convolution algorithm, calculates the execution time to equal $\Theta(|a|*|b|)$, where $|a|$ and $|b|$ are the lengths of the two input arrays to convolve. Thereby, the execution time of the implementation is approximately proportional to the product of the lengths of the two input arrays. The proportionality constant is unknown but should remain relatively constant especially for small changes in the lengths of the input arrays. From these details, it is evident that as long as the product of $|a|$ and $|b|$ is the same, the asymptotic analysis will calculate the same estimate for execution time. As a result, an effective work metric mapping is to simply group together the invocation parameters based on the product of their input array lengths, which is possible by setting the work metric value equal to $|a|*|b|$. This work metric mapping meets the two
guidelines specified in Section 4.2.1 as it coalesces invocation parameters with approximately the same execution time, and the estimated execution time mostly increases with increasing work metric.

The asymptotic analysis technique may be applied identically to many different types of implementations. As described in Table 4-1, the asymptotic analysis for a matrix multiply implementation calculates the execution time to equal $\Theta(m^* n^* p)$, where $m$, $n$, and $p$ are the lengths of the associated dimensions of the input matrices. As a result, an effective work metric mapping for a matrix multiply implementation is to set the work metric value equal to $m^* n^* p$.

The asymptotic analysis technique for determining work metric mappings is effective because it determines how to group invocation parameters with similar execution times, but any technique which adheres to the guidelines in Section 4.2.1 would suffice. For the sorting example discussed in Section 4.2.1, the work metric mapping was to set the work metric value equal to the number of elements to sort, but this is not the same as the result returned by the asymptotic analysis. Depending on which sorting algorithm the implementation uses, the asymptotic analysis may calculate the execution time to equal $\Theta(n^* \log(n))$ for a quick sort implementation, where $n$ is the number of elements to sort. While setting the work metric value equal to $n^* \log(n)$ may also be used, the asymptotic analysis reveals that the execution time is a monotonically increasing function of only $n$, and thereby invocation parameters with the same number of elements to sort will also have approximately the same execution time. As a result, setting the work metric value equal to $n$ alone is also an effective work metric mapping.
4.2.3 Design of Adapters

Table 4-2. A description of the functions required for an adapter along with example functions for a sorting adapter.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example for Sorting Adapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>RoundMetric</td>
<td>Arbitrary value passed as input and function returns the closest valid work metric value</td>
<td>Round to the nearest non-negative integer</td>
</tr>
<tr>
<td>NextMetric</td>
<td>Arbitrary value passed as input and function returns the smallest valid work metric value greater than the inputted value</td>
<td>Round to the nearest non-negative integer and add one</td>
</tr>
<tr>
<td>CalcMetric</td>
<td>Invocation parameters passed as input and function returns the corresponding work metric value</td>
<td>Return the length of the invocation parameter’s input array as the work metric value</td>
</tr>
<tr>
<td>CreateParams</td>
<td>Valid work metric value passed as input and function allocates and initializes invocation parameters representative of that work metric</td>
<td>Allocate a randomly initialized input array with length equal to the inputted work metric value</td>
</tr>
<tr>
<td>DeleteParams</td>
<td>Invocation parameters created during a prior CreateParams is passed as input, which the function will de-allocate</td>
<td>De-allocate the inputted invocation parameters allocated by CreateParams</td>
</tr>
</tbody>
</table>

As listed in Table 4-2, the adapter consists of five functions that map between the work metric interface and the implementation’s interface. The developer of an adapter would typically create the five functions after determining an appropriate work metric mapping for an implementation. The adapter may be used for a variety of purposes, but the design of the functions is motivated by the needs of the IA Heuristic, as will be evident in Section 4.3.

The first two functions specify the valid work metric values for an adapter. Typically, only some of the work metric values correspond to valid invocation parameters depending on the work metric mapping. For the sorting example discussed in Section 4.2.1, the work metric mapping was to set the work metric equal to the
number of elements to sort, inherently restricting the valid work metric values to the possible lengths of the invocation parameter’s input array, which must be a non-negative integer. The first function, called the `RoundMetric` function, rounds an arbitrary inputted value to the nearest valid work metric value. The second function, called the `NextMetric` function, returns the smallest valid work metric value that is greater than an inputted value. The IA Heuristic uses both of these functions, in combination with an overall bound on work metric values, called the `work metric range`, to determine a list of work metric values for which to analyze an implementation.

The remaining three functions map between the work metric values for an adapter and the corresponding invocation parameters for an implementation. The third function, called the `CalcMetric` function, maps invocation parameters for the implementation to a work metric value by calculating the corresponding work metric value. For the sorting example, this function would simply return the number of elements in the input array. The fourth function, called the `CreateParams` function, performs the inverse of the work metric mapping and maps a work metric value to invocation parameters by creating a representative instance of the invocation parameters corresponding to the inputted work metric value. This function typically requires allocating space for the input and output parameters and then initializing the parameters with valid values for the implementation. For the sorting example, this function might allocate an input array for the sorting implementation with a size specified by the inputted work metric value and then initialize that array with random values. As a consequence of the fourth function allocating space, the last function, called the `DeleteParams` function, then de-allocates any space for the inputted invocation parameters, assuming they were created during a previous
CreateParams function. The IA Heuristic uses all of the adapter's functions to interface with an implementation.

4.2.4 Limitations

The main limitation of implementation abstraction is the requirement for an effective work metric mapping. A work metric mapping essentially extracts a measure of computational work from the invocation parameters to coalesce invocation parameters with similar execution times, as discussed in Section 4.2.1. However, many implementations exhibit a large or unpredictable variance for execution times reducing the ability of any analysis on the invocation parameters alone to estimate the amount of computational work. The main effect of a less-accurate work metric mapping is a larger variance in the execution times of invocation parameters within the same work metric group, which reduces the resulting accuracy of the IPG. For the extreme case of an implementation having no predictable relationship between its invocation parameters and execution time, no valid work metric mapping can be created and, likewise, implementation abstraction would not be effective for that implementation. Despite this limitation, many implementations do allow for an effective work metric mapping. The technique discussed in Section 4.2.2 bases the mapping on the results of an asymptotic analysis, which is widely applicable to many implementations. Additionally, the results in Section 6.2 demonstrate the resulting accuracy of the IA Heuristic on several standard implementations. Even for those implementations with less-accurate work metric mappings, the IA Heuristic would still work correctly albeit with less-accurate results.

Elastic Functions also help improve the accuracy of the work metric mapping by providing usage assumptions, discussed in Section 3.3.3, to separate cases that exhibit different executing characteristics. The forward and inverse work metric mappings both
implicitly require knowledge of the common-case for the implementation’s executing characteristics. For the sorting example discussed in Section 4.2.1, the execution time required to sort an input array may vary significantly depending on whether the input array consists of randomly distributed or mostly sorted input values. Likewise, the inverse mapping (i.e., from work metric to an instance of invocation parameters) may choose a different instance depending on whether the adapter assumes the common-case consists of randomly distributed or mostly sorted input values. In general, the issue is that some implementations have a large complex invocation parameter-space for which it is difficult to create a work metric mapping that remains accurate over all possible invocation parameters, especially if the adapter assumes the common-case characteristics adhere to one pattern of invocation parameters when in fact they adhere to another. Usage assumptions help overcome this problem by specifying more information about the invocation. For the sorting example, separate usage assumptions for the randomly distributed and mostly sorted invocation characteristics would allow the user of the heuristic to select the most appropriate IPG based on their knowledge of the actual characteristics of the invocations. Having multiple usage assumptions has the effect of partitioning the invocation parameter-space such that each adapter only needs to remain accurate for a smaller subset of invocation parameters. The end result is both an easier to create work metric mapping and a more accurate IPG.

4.3 Implementation Assessment with the IA Heuristic

The IA Heuristic takes an implementation and its adapter as an input and performs an analysis, consisting of sampling and several other steps discussed later, to create an implementation performance graph (IPG). As discussed in Section 4.2, the adapter abstracts many of the details of the implementation and allows the heuristic to analyze
the implementation using the work metric interface. The heuristic operates internally only with the work metric values, which the adapter then maps to and from the implementation’s interface. Through a combination of measuring the execution time at different work metric values and statistical analyses, the IA Heuristic establishes the relationship between work metric and execution time, which it then uses to create the IPG. Lastly, the IA Heuristic saves the IPG for use by tools (or users) to estimate the execution time of the implementation for future invocations.

Figure 4-3. An example lookup into an IPG for a sorting implementation.

The IPG may be visualized as a two-dimensional piece-wise linear graph specifying the relationship between work metric and execution time, as illustrated in Figure 4-3. The IPG is stored efficiently both in-memory and on-disk as the ordered set of points describing the end-points and intersection-points of the segments. For lookups of work metric values residing between two points of the graph, the IPG uses linear interpolation to calculate the corresponding execution time. Additionally, storing the points in ascending work metric order allows for efficient binary searches, requiring only logarithmic operations (i.e., O(log n), where n is the number of points) to perform a lookup for a work metric value.

The IA Heuristic creates an IPG for an implementation by iterating over several steps that incrementally grow the IPG from the smallest to the largest valid work metric
value with the bounds specified by the *work metric range* discussed in Section 4.2.3. The heuristic grows an IPG by incrementally adding segments that effectively approximate the relationship between work metric and execution time for the duration of that segment. The point located at the right endpoint of the right-most currently established segment (or the lower bound of the work metric range if there are no known segments), referred to as the *frontier point*, will thereby always progress to the right. Once the frontier point moves all the way to the upper bound of the work metric range, which corresponds to the IPG having a segment for all valid work metric values, the IA Heuristic is complete and saves the IPG as the output of the heuristic.

![Diagram](image.png)

Figure 4-4. An illustration of the four steps of the IA Heuristic.

The IA Heuristic consists of four steps, as illustrated in Figure 4-4. The first step, called *sample collection*, measures the actual execution time of the implementation invoked with invocation parameters at a determined work metric value, as discussed in Section 4.3.2. The heuristic collects samples to ascertain the relationship between work
metric and execution time, which the heuristic uses in the later steps when identifying new segments. The second step, called *segment identification*, collectively analyzes the samples collected during the current and previous iterations to identify linear regions which may be approximated by a segment, as discussed in Section 4.3.3. The heuristic relies on a statistical analysis of the samples in conjunction with a set of rules to establish whether a set of samples are sufficiently linear. Typically, the heuristic will identify several candidate segments during the segment identification step, which the third step, called *segment insertion*, will then compare to determine which segment to actually insert into the IPG, as discussed in Section 4.3.4. The heuristic uses a set of criteria to compare the candidate segments and determine which is the best addition to the IPG overall. When the heuristic inserts a segment into the IPG, the new segment may possibly intersect or overlap a previously inserted segment, requiring the heuristic to then also adjust those segments. A heuristic parameter, called the *active segment window*, specifies how many of the right-most segments allow adjustments. The fourth step, called *segment commitment*, prevents further adjustments to segments once they are no longer within the active segment window, as discussed in Section 4.3.5. All four steps repeat each iteration until the heuristic completes.
Table 4-3. A description of the IA Heuristic’s tunable parameters for controlling tradeoffs between performance and accuracy.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Segment Window</td>
<td>The number of right-most segments that allow adjustments when creating an IPG. Once a segment leaves the active segment window, it becomes committed.</td>
</tr>
<tr>
<td>Sampling Execution Time Spacing</td>
<td>The desired execution time spacing between samples expressed as both a percentage increase and with optional absolute bounds on the minimum and maximum increase. The IA Heuristic uses the portion of the IPG generated so far to extrapolate the work metric to sample next based on this parameter.</td>
</tr>
<tr>
<td>Maximum Execution Time Spacing</td>
<td>The maximum execution time spacing allowed between samples expressed as both a percentage increase and with optional bounds on the minimum and maximum increase. Only segments which meet this requirement will become candidate segments for the IA Heuristic to insert into the IPG.</td>
</tr>
<tr>
<td>Segment Confidence Threshold</td>
<td>The maximum allowed width of the confidence interval calculated at the segment’s endpoints. A wider confidence interval reflects a larger variance of the samples around the linear regression.</td>
</tr>
<tr>
<td>Maximum Point Samples</td>
<td>The maximum number of samples the IA Heuristic may collect at a single work metric value before it automatically becomes a candidate segment despite the variance of the samples.</td>
</tr>
<tr>
<td>Sample Error Threshold</td>
<td>The maximum allowed error between a sample’s measured execution time and the execution time estimated by the IPG before the IA Heuristic includes the sample as part of the comparison criteria during segment insertion.</td>
</tr>
</tbody>
</table>

The IA Heuristic incorporates several tunable parameters for controlling tradeoffs between performance and accuracy, as listed in Table 4-3. These parameters will typically be hard-coded by the user of the heuristic with values appropriate for the accuracy requirements of the resulting IPG. Most of the parameters are specific to individual steps of the heuristic and, as a result, discussions of the individual parameters are deferred until their respective contexts.

4.3.1 Performance/Accuracy Tradeoff

The design of the IA Heuristic must balance two competing goals. On one hand, the heuristic should collect as few samples as possible because each sample requires
the execution of an implementation and correspondingly lengthens the time required for
the heuristic to complete. On the other hand, each sample the heuristic collects reveals
more information about the relationship between work metric and execution time,
requiring the heuristic to collect more samples to improve the accuracy of the IPG. As
the IA Heuristic desires to both complete in a reasonable amount of time and result in
an IPG of useable accuracy, the heuristic relies on an adaptive approach.

The IA Heuristic adapts to an implementation by having the user of the heuristic
specify the desired accuracy of the resulting IPG. The accuracy of an IPG is measured
as the maximum amount of discrepancy between the actual execution time of an
implementation and the execution time estimated by the IPG. As execution times may
have a large range (e.g., from microseconds to days), the desired accuracy level is
specified as an allowable error percentage in addition to optional bounds on the
minimum and maximum absolute error. For example, a user may specify the IPG should
be accurate within 5% with a lower bound of 10ms and an upper bound of 1 minute. The
minimum error bound is useful as small execution times (e.g., sorting five elements
takes only microseconds) might otherwise have impractically small error percentages
that are less than the variance or noise of the time measurements. The maximum error
bound may be useful if the user requires the IPG to have a bounded maximum error.

The only assumptions the IA Heuristic may make on the relationship between work
metric and execution time are the two guidelines for the adapter as described in Section
4.2.1. The first guideline requires invocation parameters that map to the same work
metric value to have approximately the same execution time. As an IPG essentially
maps work metric values to execution times, the applicability of the IPG to the various
possible invocation parameters is significantly dependent on how well the adapter adheres to this guideline. The second guideline is that the execution time should be non-decreasing for increasing work metric values. This guideline allows for the heuristic to make assumptions on the range of execution times for work metric values between collected samples, and thereby allows the opportunity for the heuristic to collect fewer samples while still meeting accuracy requirements.

From the assumption that execution time does not decrease as work metric increases, the IA Heuristic can establish bounds on the maximum error of estimated execution times between samples. For example, if the execution time at work metric $w_1$ is $e_1$ and the execution time at work metric $w_2$ is $e_2$, then all of the execution times for work metrics between $w_1$ and $w_2$ must be bounded by $e_1$ and $e_2$ (because otherwise the execution time would have had to decrease at some point). Likewise, if the heuristic relies on linear interpolation to approximate the execution time between the work metrics $w_1$ and $w_2$, the maximum error between the actual and estimated (i.e., interpolated) execution time would have to be less-than $e_2 - e_1$ and with a percentage less-than $(e_2 - e_1) / e_1$. From this reasoning, as long as the IA Heuristic ensures all pairs of sequential samples meet a maximum execution time spacing (METS) within the percentage (and absolute bounds) of the desired accuracy, referred to as the METS technique, the entire IPG will meet the accuracy requirements.

The IA Heuristic relies on a combination of the METS technique and statistical analysis to efficiently create an IPG with a desired level of accuracy. The heuristic uses the METS technique, as discussed previously, to limit the maximum error of the IPG.
The statistical analysis then analyzes sets of sequential samples to identify linear regions from which to create the segments for the IPG.

4.3.2 Sample Collection Step

The sample collection step of the IA Heuristic occurs first in every iteration and collects a single sample for the latter steps of the iteration to analyze. While segment collection does not, itself, analyze the samples, it is the only step which directly interacts with the implementation (through the adapter) and therefore has a goal to collect samples which enable the latter steps to progress the frontier point and create more of the IPG.

Sample collection operates by performing several sub-steps. First, sample collection selects a work metric value using the procedure discussed later in this section. Second, sample collection uses the adapter’s `RoundMetric` function to round the selected work metric value to the nearest valid value. Third, sample collection uses the adapter’s `CreateParams` function to create an instance of invocation parameters representative of the selected work metric value. Fourth, sample collection executes the implementation using the created invocation parameters and measures the execution time required for the implementation to complete. Lastly, once the implementation completes, sample collection uses the adapter’s `DeleteParams` function to de-allocate the created invocation parameters. The combination of selected work metric value and resulting execution time is then the newest sample for the latter steps of the current iteration of the heuristic to process. The details of the adapter's functions are discussed in Section 4.2.3. The remainder of this section discusses the selection of the work metric value.
The only way for sample collection to select a new work metric value is to extrapolate based on the previously collected samples. For this purpose, sample collection analyzes the portion of the IPG created up through the end of the preceding iteration. Specifically, sample collection takes into account the current location of the frontier point and the slope of the right-most segment. Note that sample collection bases the selection of the new work metric value only on the IPG and not directly on the previously collected samples. As the segment identification step creates segments in the IPG only when sets of sequential samples meet the set of criteria described in Section 4.3.3, not every collected sample will immediately result in a segment in the next iteration’s IPG.

The procedure to select a new work metric value is based on how many points and segments are in the IPG. For the first few iterations when no points are in the IPG, sample collection selects the work metric value corresponding to the lower bound of the work metric range to establish the left-most point of the IPG. Once the IPG comprises of only a single point (which must be at the lower bound of the work metric range), no slope information is yet available, so sample collection uses the adapter’s NextMetric function to select the work metric value immediately adjacent to the lower bound of the work metric range. Once the IPG comprises of at least two points, slope information for the right-most segment is available and sample collection can extrapolate the location of work metric values based on a desired execution time spacing, allowing for usage of the METS technique.
Figure 4.5. An illustration of sample collection determining a work metric value to sample.

Following the rationale of the METS technique, discussed in Section 4.3.1, sample collection attempts to extrapolate the location of the work metric value with an execution time spacing specified by a heuristic parameter called the *sampling execution time spacing* (SETS) parameter. As illustrated in Figure 4-5, sample collection assumes the slope of the right-most segment remains approximately consistent through the yet to be collected sample. Sample collection then takes the inverse slope of the right-most segment (i.e., change in work metric divided by change in execution time) and multiplies it by the desired increase in execution time to calculate how much to increase the work metric beyond the work metric of the frontier point for the next sample. The SETS parameter specifies the execution time spacing both as a percentage increase and with optional absolute bounds on the minimum and maximum spacing. Regardless of whether or not the new sample’s execution time is within the spacing (i.e., the slope is not consistent through the collected sample), sample collection will still collect the sample and give the remaining steps in the iteration the opportunity to utilize the new sample (in conjunction with the previously collected samples) to identify new segments. As one of the criteria for identifying a segment is to verify that the actual execution time spacing between samples is within the accuracy requirements, the remaining steps may
not identify a new segment and, likewise, the frontier point will not progress. Following which, the next iteration will start once again without any change to the IPG.

When the sample collection step starts without any progression of the frontier point since the previous iteration, sample collection uses an exponential fallback technique to select work metric values closer and closer to the frontier point. As illustrated in Figure 4-5, the exponential fallback technique sets the new work metric value equal to the value halfway between the work metric value sampled during the previous iteration and the minimum sampling work metric (MSWM) value, which is the smallest valid work metric value greater than the work metric value of the frontier point as calculated by the adapter’s NextMetric function. The exponential fallback technique ensures that when the IPG does not progress, the next iteration will collect a sample with a closer work metric spacing with the frontier point to encourage the identification of new segments. While not all of the samples will immediately result in the identification of a new segment, none of the samples go unused. All collected samples will eventually form the basis of a segment, whether for the current or a later iteration.

In addition to the previous technique to select a new work metric value, sample collection also performs several validity checks to prevent potential complications. First, sample collection clips the work metric value to be within the bounds of the work metric range. Second, sample collection clips the work metric value such that the work metric spacing between the sample to collect and the frontier point is no greater than the work metric spacing between the frontier point and the lower bound of the work metric range, essentially limiting the growth of the IPG so that it cannot more than double in size each iteration. This restriction is necessary to prevent the problem of a few closely spaced
noisy samples in the first few iterations from resulting in a nearly horizontal segment, and therefore forcing sample collection to sample at a work metric value equal to the upper bound of the work metric range (because a nearly horizontal segment would require a large work metric increase to accomplish even a small increase in execution time). Lastly, sample collection verifies that the exponential fallback technique always selects a work metric value closer to the frontier point than the previous iteration, until it reaches the MSWM value. Without this check, rounding issues (i.e., with the \textit{RoundMetric} function) may result in the exponential fallback technique never reaching back to the MSWM value.

\subsection*{4.3.3 Segment Identification Step}

The segment identification step of the IA Heuristic occurs second in every iteration and analyzes the latest sample, in conjunction with the samples collected from previous iterations, to identify when sets of sequential samples may be represented as a segment in the IPG. Segment identification uses a statistical analysis and a set of criteria to determine if a set of samples are sufficiently linear to be represented as a segment. Typically, the step will identify several candidate segments from which the latter steps will then compare to determine which to insert into the IPG.

Segment identification analyzes sets of samples using a linear regression analysis. A linear regression analysis is a standard statistical technique that processes a set of samples and calculates the line that minimizes the mean-squared distance between the samples and the line. The analysis also quantifies how accurately the line approximates the samples by calculating a confidence interval, which is another standard statistical technique based on the variances of the samples around the line.
As neither the starting nor ending points of the segments are known beforehand, the segment identification step performs a linear regression analysis on all sequential subsets of the samples. For this purpose, segment identification relies on a data structure, illustrated in Figure 4-6, called the *regression matrix*. The regression matrix may be visualized as an upper-right triangular matrix whose rows correspond to different starting samples for the linear regression analysis and columns correspond to different ending samples. Indexes in the matrix refer to the indexes of the samples in increasing work metric order. Likewise, every cell in the matrix stores the result of a linear regression analysis performed on a distinct subset of the sequential samples. The work metric values of the first and last samples of the subset correspond to the interval of work metric values for which the analysis is pertinent. For example, the result of the linear regression analysis on the subset of sequential samples indexed two through four (in increasing work metric order) would be located at the cell in row two, column four. Likewise, if the work metric values of samples indexed two and four were 35 and 110, the corresponding linear regression analysis would pertain to the work metric interval of 35 through 110. The matrix is upper-right triangular in shape as the starting index cannot be greater than the ending index.
When the segment identification step starts, it must first incorporate the newly collected sample for the current iteration into the regression matrix. As illustrated in Figure 4-7, incorporating a sample starts by finding the sorted index (in increasing work metric order) of the new sample among the previously collected samples. If the new sample’s work metric value is unique (i.e., this is the first sample with this work metric value), segment identification will then insert a new row and column into the regression matrix at the position corresponding to the index of the new sample. If the new sample’s work metric value is not unique (which is possible due to the exponential fallback technique in sample collection), the step will instead accumulate the statistics of the new sample with the previous samples of the same work metric value at their corresponding cell. Segment identification then performs a linear regression analysis on all intervals that include the new sample. In order for an interval to include the new sample, its starting index (i.e., its row) must be less than or equal to the index of the new sample and its ending index (i.e., its column) must be greater than or equal to the index of the new sample, which corresponds to a rectangular region of cells in the upper-right portion of the regression matrix. Segment identification may simply copy
over the cells outside of this region as their subsets of samples do not include the new sample and thereby their linear regressions analyses would not have changed from the previous iteration.

The segment identification step populates the cells in the regression matrix efficiently by using a dynamic programming algorithm. Performing a linear regression analysis requires only the accumulation of simple calculations on the coordinates of the samples (i.e., accumulating the $x$, $x^2$, $y$, $y^2$, and $x \cdot y$ values where $x$ is the work metric of the sample and $y$ is the execution time). As a result, a linear regression analysis on a large number of samples may be performed instead by first dividing the samples into subsets, calculating the partial accumulations for each subset individually, and then accumulating the partial sums when performing the overall linear regression analysis. Equivalently, segment identification performs the linear regression analyses in the order of increasing number of samples and saves the partial accumulations for each analysis, so that populating a cell in the regression matrix requires only accumulating the saved partial sums from the prior analyses corresponding to the two halves of the set of samples. Segment identification populates all cells in the regression matrix in this manner with the exception of the cells along the regression matrix’s diagonal, which correspond to the linear regression analyses at a single work metric value and thereby cannot be further subdivided. Segment identification instead populates these cells directly by performing the linear regression analyses on the samples themselves, which then form the base cases for populating the remaining cells.
After populating the cells of the regression matrix with linear regression results, the segment identification step must then analyze the cells to identify candidate segments. Segment identification creates a segment for a cell by taking the linear regression result in the cell (which corresponds to a line) and clipping it to the interval of work metric values pertaining to the cell (i.e., the starting and ending work metric values for the analysis’ subset of samples). Segment identification only considers segments that will progress the frontier point and not leave any work metric discontinuities in the IPG, which corresponds to cells with a starting work metric (i.e., row) of less than or equal to the MSWM value and an ending work metric (i.e., column) of greater than or equal to the MSWM value. Note that even a segment which starts at the MSWM value will not create a discontinuity as there are no valid work metric values between the MSWM value and the work metric of the frontier point. In the regression matrix, this corresponds to rectangular region of cells, as illustrated in Figure 4-8.

Segment identification then analyzes the cells and relies on several requirements to determine whether or not the samples are sufficiently linear to become a candidate segment. First, the linear regression analysis must comprise of at least three samples so that the confidence interval results are meaningful. Second, the two end-points of the segment must have positive execution time values. Third, the density of samples over the work metric interval of the segment must be sufficient to guarantee the accuracy of
the analysis along the entire interval. Following the rationale of the METS technique, segment identification requires the samples to be within a maximum spacing as specified by a heuristic parameter called the \textit{maximum execution time spacing} (METS) parameter. Specifically, segment identification takes the inverse slope of the segment (i.e., change in work metric divided by change in execution time) and multiplies it by the maximum allowable increase in execution time to calculate the maximum allowable work metric spacing between samples. The METS parameter specifies the maximum execution time spacing both as a percentage increase and with optional absolute bounds on the minimum and maximum spacing. Lastly, segment identification calculates the confidence interval for the two endpoints of the segment to verify that the segment accurately approximates the samples. A larger confidence interval corresponds to the samples having more variance around the segment. Another heuristic parameter, called the \textit{segment confidence threshold} parameter, specifies the maximum allowed width of the confidence interval. If the segment meets all of these criteria, it then becomes a candidate segment.

In addition to the previous requirements, the segment identification step also supports two corner-cases to prevent deadlock in certain situations. First, segment identification guarantees that the spacing required to meet the METS parameter is attainable by the current adapter. If it cannot make this guarantee, it rounds up the maximum spacing to the next valid work metric value using the adapter’s \texttt{NextMetric} function. Second, segment identification ignores the confidence interval requirement when the segment is zero-length (i.e., is a point) and consists of more than a certain number of samples, as specified by a heuristic parameter called the \textit{maximum point}
samples parameter. Otherwise, an implementation with a large variance around a specific work metric value may never meet the confidence interval requirement resulting in deadlock of the heuristic.

4.3.4 Segment Insertion Step

The segment insertion step of the IA Heuristic occurs third in every iteration and compares the candidate segments to determine which is the best overall addition to the IPG. The checks performed by the segment identification step already have verified that the candidate segments accurately estimate the samples for different intervals of work metric values, but the segment insertion step further checks which segment results in a smooth and accurate IPG of minimum number of segments. Segment insertion relies on a set of comparison criteria to analyze each candidate segment and determine which to insert. As part of inserting a new segment, the segment insertion step may also need to resize or remove segments previously inserted into the IPG to accommodate the new segment.

![Diagram showing three possible insertion locations for a segment]

Figure 4-9. An example of the three possible insertion locations for a segment.

Each candidate segment may have up to three types of insertion locations into the IPG based on the locations of segments already in the IPG, as illustrated in Figure 4-9. First, if the candidate segment starts at the MSWM value, then the only possible insertion location is to append the segment (unchanged) into the IPG. Second, if the
candidate segment starts prior to the MSWM value and intersects one or more of the already present segments of the IPG, then each intersection point is another possible insertion location. Inserting a segment at an intersection point also requires accordingly resizing the candidate segment to start at the intersection point and resizing/removing the already present segments to end at the intersection point. Third, if the candidate segment starts at the lower bound of the work metric range, then another possible insertion location is to remove all of the existing segments and replace the entire IPG with the candidate segment. Note that segment insertion considers only these insertion locations to guarantee that the IPG will have a segment for every valid work metric value and to minimize the number of discontinuities (i.e., jumps in execution time between segments). Note that inserting a segment that starts at the MSWM value may result in a jump in execution time in the IPG, but does not leave any work metric values without a corresponding segment as no valid work metric values exist between the MSWM value and the work metric value of the frontier point.

Segment insertion individually compares the IPGs that would result from inserting each candidate segment at each of its possible insertion locations to determine which segment and insertion location is the overall best, as determined by the set of criteria below. First, the IPG that results in the fewest number of samples with significant error is better. Segment insertion considers a sample to have significant error if the difference between the sample’s execution time and the execution time estimated by the IPG is larger than a threshold specified by a heuristic parameter called the sample error threshold (SET) parameter. The SET parameter is specified both as an error percentage and with optional absolute bounds on the minimum and maximum allowed...
error. Second, for IPGs with equal number of samples with significant error, the IPG that ends at a larger work metric value is better. Third, the IPG with the least number of segments is better. Lastly, for IPGs that are equal for all of the previous criteria, the IPG with the lowest mean-squared-error percentage between the samples’ execution times and the execution time estimated by the IPG is better. Segment insertion then inserts the winning segment into the IPG for the next step of the IA Heuristic.

4.3.5 Segment Commitment Step

The segment commitment step occurs last in every iteration and commits segments once they move a specified number of segments to the left of the frontier point. Once a segment is committed, segment insertion may no longer resize or remove the segment as part of the process of inserting a new candidate segment into the IPG. Additionally, segment commitment removes the segment’s corresponding samples from the regression matrix, reducing memory overhead and improving efficiency for latter iterations.

![Figure 4-10. An illustration of removing the top rows of the regression matrix during segment commitment.](image)

After several iterations, the number of samples in the regression matrix may get large. Correspondingly, the segment identification step, which analyzes the cells in the regression matrix to identify candidate segments, will require more processing overhead to analyze the greater number of potential starting and ending locations for the
candidate segments. To prevent this overhead from becoming prohibitive, the segment commitment step gradually removes rows from the top of the regression matrix corresponding to starting locations far to the left of the frontier point. As the row index of the regression matrix corresponds to the starting sample of the corresponding linear regression analysis (indexed in increasing work metric order), removing an entire row from the top of the regression matrix is equivalent to preventing segment identification from considering segments which start at the corresponding sample’s work metric value. As illustrated in Figure 4-10, segment commitment removes all of the top rows corresponding to starting locations within the work metric interval of any segment that moves more than a certain number of segments to the left of the frontier point, with the number of segments specified by a parameter called the *active segment window* parameter. In other words, only the segments within the active segment window allow changes because segment identification can only consider starting locations that correspond to cells in the regression matrix, and segment commitment removes all of the cells corresponding to starting location that would overlap segments not in the active segment window.

All four steps of the IA Heuristic repeat and continue to progress the frontier point – with segment collection collecting segments after the frontier point, segment identification identifying candidate segments that may connect with the segments in the active segment window, segment insertion inserting only the best candidate segment into the IPG, and segment commitment eventually preventing further changes to segments far to the left of the frontier point – until the frontier point finally progresses to the upper-bound of the work metric range. At this point, a segment of the IPG will exist
to estimate the execution time for any valid work metric value within the work metric range of the IPG and the IA Heuristic is complete.

4.4 Summary

In this chapter, we described how Elastic Computing abstracts implementations and performs the implementation assessment step. Elastic Computing requires implementation abstraction to hide many of the implementation-specific details that would otherwise prevent Elastic Computing from supporting nearly any kind of computation. The elastic function supports abstraction by means of a developer-created abstraction layer, called an adapter, which maps the input parameters to an abstract quantity called a work metric. The work metric interface is the only interface the implementation assessment and optimization planning steps use to communicate with an implementation, allowing Elastic Computing to support any elastic function that has an associated adapter.

Using the adapter, the implementation assessment step then creates a performance estimating data structure, called an implementation performance graph (IPG), to estimate the execution time of an implementation from its input parameters. The structure of the IPG is a piece-wise linear graph data structure that maps work metric (calculable from the input parameters via the adapter) to the implementation’s estimated execution time. A separate IPG is created for each implementation executing in different execution situations.

Chapter 5 continues on by describing parallelizing templates and the details of the optimization planning step, which internally relies on implementation assessment, and ends with a discussion on elastic function execution. Lastly, Chapter 6 presents
accuracy results for implementation assessment and speedup results achieved by Elastic Computing for several elastic functions.
CHAPTER 5
ELASTIC FUNCTION PARALLELIZATION, OPTIMIZATION, AND EXECUTION

In this chapter, we describe how Elastic Computing parallelizes, optimizes, and executes elastic functions. Section 5.1 provides an overview of the optimizing and executing process. Section 5.2 describes the structure of parallelizing templates, which allow developers of elastic functions to specify how Elastic Computing may parallelize computation. Section 5.3 then describes the RACECAR heuristic, which is the algorithm Elastic Computing uses to perform optimization planning. Lastly, Section 5.4 describes how Elastic Computing executes an elastic function at run-time by referring to the results of the RACECAR heuristic. The speedup results achieved by Elastic Computing and the RACECAR heuristic are presented in Chapter 6.

5.1 Overview

Determining efficient elastic function execution decisions, also called optimization planning, is the second of the two installation-time optimization steps Elastic Computing performs. The first step, called implementation assessment, creates performance estimating data structures, called implementation performance graphs (IPGs), for each of the implementations of an elastic function, as discussed in Chapter 4. From the IPGs, optimization planning then compares the execution times of alternate execution options to determine execution decisions that minimize execution time. Optimization planning then saves the results for Elastic Computing to refer to at run-time when an application invokes an elastic function.

The goal of optimization planning is to answer two main questions. First, when an application invokes an elastic function, which implementation and resources will be the most efficient for the current executing situation? Note that the answer to this question
changes based on the actual invocation parameters and the resources available for execution. Second, when a parallelizing template is executed, what is the most efficient partitioning of computation and resources? Optimization planning answers both questions by reinterpreting the questions as optimization problems with the goal being to minimize the execution time as estimated by the IPGs.

5.2 Parallelizing Templates

Parallelizing templates provide Elastic Computing with knowledge of how it may parallelize the computation of an elastic function. Developers create parallelizing templates identically to how they create implementations, and similarly incorporate the templates as part of an elastic function. Unlike implementations, parallelizing templates are not an independent execution alternative of an elastic function, but instead specify how Elastic Computing may partition the computation of an elastic function invocation into two smaller, independent elastic function invocations. Transforming one big elastic function execution into two smaller elastic function executions allows Elastic Computing to then execute the smaller invocations in parallel on distinct subsets of system resources, improving the overall execution time by increasing the amount of parallelism. While each parallelizing template only partitions computation into two, Elastic Computing may nest the parallelizing templates arbitrarily deep, allowing for the creation of as many partitions of computation as continue to improve performance, utilizing up to all of the resources on a multi-core heterogeneous system.
The basic structure of a parallelizing template mimics the design of a divide-and-conquer algorithm, as illustrated in Figure 5-1 for a merge sort parallelizing template. For any function which supports a divide-and-conquer algorithm (e.g., merge-sort for a sorting elastic function), a developer may create a corresponding parallelizing template. The structure of the parallelizing template is to first partition the input parameters into two smaller sets of input parameters, invoke two nested instances of the elastic function to calculate the output for the two smaller sets of input parameters, and then merge the outputs of the two nested elastic functions when complete. The answer for what resources and implementations to use for each nested elastic function as well as how much of the computation to partition to each call is determined by the RACECAR heuristic, as discussed in Section 5.3.

5.3 Optimization Planning with the RACECAR Heuristic

RACECAR is a heuristic that determines efficient execution decisions when optimizing elastic functions onto a multi-core heterogeneous system. The inputs to the heuristic are a set of implementations, parallelizing templates, and additional information describing the function and the resources available on the system. Note that
the heuristic is not required to know the specific input parameters for the function beforehand, as the heuristic instead determines efficient ways to execute the function for all possible input parameters. The operation of the heuristic, discussed later, then analyzes the performances of the different implementations and iteratively builds data structures that specify how to efficiently execute the function for any set of input parameters. The heuristic saves the resulting data structures for reference at run-time when an application invokes an elastic function.

The output of RACECAR is two data structures called the *implementation table* and the *parallelization table*. The implementation table, when provided with input parameters and an optional restriction on allocated system resources, returns the implementation that RACECAR estimates to provide the minimum execution time. Any executing instances of parallelizing templates may then also refer to the parallelization table to determine how to efficiently partition their computation.

Figure 5-2. The high-level steps of the RACECAR heuristic.
RACECAR consists of six iterative steps, illustrated in Figure 5-2, that correspond to letters in the RACECAR acronym – Recursive Add, Compare, Execute, Compare, Apply, and Repeat. These steps are listed in order, but the first two steps only occur after the first iteration, and therefore are not described until after the first four steps. For each iteration, the heuristic optimizes for a specified subset of the system’s resources, called the iteration’s working set. Initially, the working set is a single CPU, but at the end of each iteration, the heuristic gradually grows the working set until it comprises of all system resources. The “execute” step (Section 5.3.1) considers all implementations (and parallelizing templates) that may execute on the current working set and relies on the IA Heuristic (Section 4.3) to generate a performance-estimation data structure called an implementation performance graph. Following which, the “compare” step (Section 5.3.2) combines the created implementation performance graphs into a function performance graph, which stores information on only the most efficient implementations for the function. The “apply” step then saves the function performance graph into the implementation table. If the working set does not comprise of all system resources, then all six steps will “repeat” using a slightly larger working set. Using the new working set, the “recursive add” step (Section 5.3.3) then considers all possible ways to partition the resources for the current working set, looks up the corresponding function performance graphs, and performs an “adding” operation to generate a restricted parallelization graph, which informs a parallelizing template of how to partition computation for that division of resources. Following which, the “compare” step (Section 5.3.4) operates almost identically to the previous compare step, but instead combines the restricted parallelization graphs into a parallelization graph, which informs a parallelizing template
of how to both partition resources and computation to minimize execution time. Note that the parallelizing templates can only execute when there is more than a single resource to perform computation, which is why the first two steps only execute after the first iteration when the working set is larger than a single CPU. All six steps iterate until RACECAR has evaluated a working set of all resources.

5.3.1 Integration with the IA Heuristic

The “execute” step of RACECAR determines the relative performances of each implementation that is executable in the current working set by relying on the IA Heuristic (Section 4.3) to generate an implementation performance graph. Each implementation performance graph specifies the estimated execution time of an implementation for different work metrics, which are abstractions of the function’s input parameters, as described in Section 4.2. RACECAR is, by design, not specific for any type of function, and therefore can make no assumptions about the structure of the input/output parameters or the performance characteristics of the implementations. As a result, RACECAR analyzes the implementations only in terms of their work metric interfaces as provided by the implementations’ adapters (Section 4.2).

![Implementation Performance Graph for Sorting Implementation](image)

Figure 5-3. Example of an IPG for a sorting implementation. The illustrated lookup within the IPG returns the estimated execution time for the implementation to sort 10,000 elements.
An implementation performance graph, as illustrated in Figure 5-3, may be visualized as a two-dimensional piece-wise linear graph that maps work metric to execution time. Estimating the execution time of an implementation using this graph simply requires calculating the work metric associated with the input parameters and looking up the corresponding execution time for that work metric. The simplistic structure of the graph allows for very efficient lookups by performing a binary search, and supports the performing of further analyses, such as those performed by the other steps of the RACECAR heuristic.

5.3.2 Creation of Function Performance Graphs

The “compare” step of RACECAR compares the implementation performance graphs of all the implementations that are executable within the working set, and then combines them to create a *function performance graph*, which stores information about only the most efficient implementations for executing the function at different work metrics. Any implementation performance graph created during the current iteration’s “execute” step must be considered, as well as any previously generated implementation performance graphs that correspond to implementations executing on proper subsets of the working set’s resources. For example, if the current working set has 4 CPUs and an FPGA, RACECAR would consider any implementation performance graph of implementations executing on 1, 2, 3, or 4 CPUs with or without an FPGA. The order that RACECAR loops through the working set resources guarantees that it would have already iterated through all proper subsets of the working set’s resources beforehand, allowing the heuristic to simply retrieve those implementation performance graphs.
Creating a function performance graph from a set of implementation performance graphs is a simple process due to the simplicity of the data structures. As the mapping from input parameters to work metric is consistent for all of the graphs, the interpretation of the x-axis of the graphs is also consistent, and locating the best performing implementation is as simple as locating the graph with the lowest execution time at any specific work metric value. As illustrated by Figure 5-4, RACECAR performs this process for all work metric values by overlaying the implementation performance graphs and saving only the collection of segments and intersection points that trace the lowest boundary for all work metric values, which is also called the lowest envelope of the graph. To perform this process, RACECAR uses a modified Bentley-Ottmann computational geometry algorithm [4] that starts at the lowest work metric value, determines which implementation performance graph is the best performing, and then proceeds with a sweep-line that checks for when another graph might outperform the current best by testing at intersection points between the segments of the graphs. The algorithm proceeds from the smallest to the largest work metric value and saves the
collection of segments and intersection points that describe the lowest envelope, in addition to other information about which graph sourced the corresponding segment.

Unlike the implementation performance graphs, which are associated with individual implementations, the function performance graphs are associated with the elastic function. A single lookup in the function performance graph returns the most efficient implementation of the elastic function, when constrained to using any of the provided implementations that execute within the current working set’s resources. The heuristic saves the function performance graph in the “apply” step as part of the output in the implementation table.

5.3.3 Creation of Restricted Parallelization Graphs

The “recursive add” and “compare” steps of RACECAR inform the parallelizing templates of how to efficiently partition their computation across different resources by generating a parallelization graph. The parallelization graph informs the parallelizing templates of what resources to use and how much computation to allocate for each of their recursive function calls. RACECAR creates the parallelization graph in two steps. First, the “recursive add” step creates a restricted parallelization graph, which answers only the question of how much computation to apportion for a fixed partitioning of resources. Second, the “compare” step then combines the created restricted parallelization graphs for all possible resource partitions to form the parallelization graph.
Figure 5-5. Structure of a merge-sort parallelizing template illustrating computation partitioning, resource partitioning, and calculation of the overall execution time.

All parallelizing templates are assumed to adhere to the following parallelization strategy, as illustrated with a merge-sort example in Figure 5-5. The implementation must partition the input into two subsets and then perform two parallel recursive executions of the function to independently process the subsets. After the recursive executions complete, the implementation must combine the outputs to form the overall result. Divide-and-conquer, data-parallel, and other algorithms support structuring their computation in this way.

Determining the best partitioning of computation for an instance of a parallelizing template requires first specifying a computation's permitted partitionings. For most types of parallelizing templates, a valid partitioning of the computation may be specified by relating the input parameters of the two recursive function calls to the overall invocation parameters of the parallelizing template. As RACECAR uses the abstraction of a work metric, this partitioning specification can be represented as an equation, called a work metric relation, relating the work metrics of the two recursive calls to the work metric of the invocation parameters. For example, a merge-sort parallelizing template requires that the sum of the sizes of its recursive sorts equals the overall input size of the sort.
Likewise, the corresponding work metric relation would equivalently state that the sum of the recursive function call work metrics must equal the work metric of the invocation parameters, as illustrated in Figure 5-5. In fact, any parallelizing template that sets the work metric proportional to the amount of computation and follows the structure of dividing computation (without overlap) between the two recursive function calls, would also adhere to this work metric relation, making it very prevalent in common parallelizing templates. As a result, we refer to this as the standard work metric relation and assume that the parallelizing templates support this relation for the upcoming discussion. A discussion of handling other relations is presented at the end of the section.

In addition to restricting the partitioning of computation, a restriction must also be placed on the partitioning of the resources. RACECAR requires that the apportioning of resources to the recursive calls must be from the same subset of resources allocated to the overall parallelizing template. As the recursive calls must also execute simultaneously (i.e., in parallel), the two calls must therefore execute within distinct proper subsets of the parallelizing template’s resources. We refer to this restriction as the resource relation.

As illustrated in Figure 5-5, the overall execution time of a parallelizing template is the time of the partitioning and combining steps plus the maximum of the execution times of the two recursive calls. If it is assumed that the execution times of the partitioning and combining steps do not vary significantly with how the computation is parallelized, then minimizing the overall execution time is equivalent to minimizing the maximum execution time of the two recursive calls. Additionally, when an iteration of RACECAR creates a parallelization graph, the corresponding working set is the set of
resources allocated for the parallelizing template, thereby previous iterations of the heuristic would have already created function performance graphs for any proper subset of the working set’s resources, allowing the usage of the function performance graphs to create an estimate of the execution time of the recursive calls. As a result, determining efficient parallelizing decisions is an optimization problem, which we call the parallelizing optimization problem (POP). The POP problem is defined as: “given a work metric relation and resource relation for a parallelizing template, the set of execution resources allocated for that parallelizing template, function performance graphs for all proper subsets of the allocated resources, and the work metric of the input parameters, determine the apportioning of work metric and resources that adheres to the relations and minimizes the maximum execution time of the recursive function calls as specified by the function performance graphs.”

RACECAR does not solve the POP problem for individual work metrics, but instead creates the parallelization graph to inform parallelizing templates of how to partition computation for any work metric. Specifically, the parallelizing templates perform a lookup within the parallelization graph based on the invocation parameter’s work metric, which returns how to efficiently apportion the work metrics and resources between the parallelizing template’s two recursive function calls.

As mentioned previously, RACECAR simplifies the creation of the parallelization graph by answering the questions of how to apportion resources and computation in two steps. The first step answers the question of how to partition the computation for a fixed partitioning of resources by creating a restricted parallelization graph. The heuristic then creates a separate restricted parallelization graph for every possible resource
partitioning. For example, if the working set of the current iteration was 4 CPUs/FPGA, then the heuristic would create restricted parallelization graphs for 1 CPU in parallel with 3 CPUs/FPGA, 2 CPUs in parallel with 2 CPUs/FPGA, etc. After creating all of the restricted parallelization graphs, the second step then combines the restricted parallelization graphs to form the overall parallelization graph, as discussed in Section 5.3.4. The remainder of this section focuses on the creation of the restricted parallelization graphs.

For each specific partitioning of resources, RACECAR creates the restricted parallelization graph by performing several steps, as illustrated in Figure 5-6. First, the heuristic retrieves the function performance graphs corresponding to the two partitions of the specified partitioning of resources. Second, the heuristic breaks up the function performance graphs into their constituent segments. Third, the heuristic processes all possible pairings of the segments from the two graphs individually in a process called segment adding. The output of segment adding is a sub-parallelization graph, which is identical in purpose to the restricted parallelization graph, but represents the optimal solution to the POP problem given only the information provided by the pair of segments. Lastly, the heuristic then combines all of the sub-parallelization graphs to
form the restricted parallelization graph, which is the globally optimal solution for the POP problem given all of the information provided by the function performance graphs. Note that optimality is defined here only in terms of the POP problem, which specifies minimizing the maximum execution time, as estimated by the function performance graphs.

![Diagram](image)

**Figure 5-7.** Example of segment adding creating a sub-parallelization graph.

**Table 5-1.** The pointer movement rules used by segment adding.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>If both pointers are at their right endpoints, then segment adding is complete.</td>
</tr>
<tr>
<td>2</td>
<td>If only one pointer is not at its right endpoint, then move that pointer to the right.</td>
</tr>
<tr>
<td>3</td>
<td>If both pointers have negative slopes and equal y’s, then move both pointers to the right.</td>
</tr>
<tr>
<td>4</td>
<td>If the pointer with the larger y has negative slope, then move that pointer to the right.</td>
</tr>
<tr>
<td>5</td>
<td>If the pointers have unequal y’s, then the pointer with the larger y must have a positive slope (or else it would have been handled previously), so move the pointer with the smaller y to the right.</td>
</tr>
<tr>
<td>6</td>
<td>The only remaining case would be if both pointers have positive slopes and equal y’s, so move both pointers to the right.</td>
</tr>
</tbody>
</table>

Segment adding generates a sub-parallelization graph by determining the optimal way to partition computation given only the information provided by the two segments.

To understand the operation of segment adding, first assume that \( x_1, y_1, x_2, y_2, \) etc. correspond to the coordinates of the endpoints of the two segments, as specified by Figure 5-7. By applying the standard work metric relation, those two segments can only
create valid partitions for computation with work metrics ranging from $x_1 + x_2$ to $x_3 + x_4$.

Likewise, the optimal way (as it is the only way) to partition the computation with a work metric of $x_1 + x_2$ would be to partition the computation into recursive calls with work metrics of $x_1$ and $x_2$ respectively, resulting in an estimated execution time of $\max(y_1, y_2)$. Similarly, the same fact applies to the two opposite endpoints of the segments with an execution time of $\max(y_3, y_4)$ and work metrics of $x_3$ and $x_4$ being the optimal way to partition computation that has a work metric of $x_3 + x_4$. Optimally partitioning the work metrics between the two endpoints may be described by first visualizing two fictitious pointers that initially start at the left endpoints of the two segments and then trace the sub-parallelization graph as the pointers move towards the right endpoints of their segments. Specifically, if the pointers are assumed to have coordinates $(x_1', y_1')$ and $(x_2', y_2')$, then the corresponding point in the sub-parallelization graph would have coordinates with a work metric of $x_1' + x_2'$ and an execution time of $\max(y_1', y_2')$. As the pointers start at the left endpoint of the two segments, they are initially optimal, but then the problem becomes determining which of either (or both) of the pointers to move towards the right at each step in such a way to preserve optimality. As the POP problem requires minimizing the maximum estimated execution time, the pointers undertake movements to first lower the execution time of the sub-parallelization graph as early as possible and then postpone increasing the execution time of the parallelization graph until as late as possible. All possible cases for how to move the pointers are listed in Table 5-1, with an example illustrated in Figure 5-7. Once both pointers reach the right endpoints of their respective segments, this situation also corresponds to the right
endpoint of the sub-parallelization graph, and therefore the entire sub-parallelization graph would have been specified during the movements of the pointers.

An informal proof of optimality may be made by induction by noting that the pointers are initially at an optimal partitioning of the computation and every move of the pointers preserves this optimality, therefore the overall result is also optimal. Note that simply applying the movements described by Table 5-1 on the two function performance graphs themselves would not guarantee optimality, as the process requires that the execution time changes consistently with the movement of the pointers (i.e., that the slope of the graphs do not change signs).

As segment adding works by moving pointers through two straight segments, the resulting sub-parallelization graph is also a piece-wise linear graph, allowing for efficient storage and processing. Additionally, the movements of the pointers through the segments do not require a complex continuous movement, but may instead be implemented by only considering when the movements of the pointers should change. Likewise, each individual movement would correspond to the creation of a new segment in the sub-parallelization graph, and allows for an efficient implementation of the segment adding process.

The sub-parallelization graphs may be visualized as a two-dimensional graph relating the work metric of the invocation parameters to the maximum execution time of the recursive calls. Additional information about the work metrics of the recursive calls, the execution time of each call, and the implementations to use for each call may also be saved in the graph, as provided by the respective segments used during the segment adding process. As the x-axes are consistent for all of the sub-parallelization
graphs, the execution times may easily be compared across different graphs to determine which parallelization results in the minimum estimated execution time. As a result, RACECAR may combine the sub-parallelization graphs in a process identical to the “compare” step described in Section 5.3.2. Likewise, as each sub-parallelization graph specifies the optimal partitioning of computation restricted to the information of only two segments, and the heuristic combines sub-parallelization graphs for all possible pairings of the segments, the resulting restricted parallelization graph is also optimal.

The previous discussion assumes that the standard work metric relation applies to the parallelizing template. The more general case of simply a linear relationship between the work metrics of the recursive calls and the invocation parameters would represent a work metric relation in the form \( C_1\alpha + C_1\beta + C_2 = x \), where \( C_1 \) and \( C_2 \) are restricted to being constant for a specified invocation parameter’s work metric and \( \alpha \) and \( \beta \) represent the work metric values of the two nested function calls, as illustrated in Figure 5-5. Note that this form handles cases where the computation may overlap between the two recursive calls. Even in this case, creating the restricted parallelization graph using the standard work metric relation still applies, as the parallelizing templates can instead integrate the constants into their lookups within the graph, corresponding to a lookup of \( y = (x - C_2) / C_1 \). More complicated non-linear forms of the work metric relation are currently not supported by the RACECAR heuristic, but are planned for future work.

5.3.4 Creation of Parallelization Graphs

The “compare” step of RACECAR considers all of the restricted parallelization graphs, corresponding to the optimal parallelization performance of different fixed partitionings of resources, and then combines them to create the parallelization graph.
As the x-axes are consistent for all restricted parallelization graphs, execution times may be compared between the different graphs to determine which partitioning of computation and resources results in the minimum estimated execution time. As a result, the heuristic combines the restricted parallelization graphs in a process identical to the previous “compare” step described in Section 5.3.2. Likewise, as each restricted parallelization graph specifies the optimal partitioning for a specific partitioning of resources, and the heuristic combines restricted parallelization graphs corresponding to all possible resource partitionings, the resulting parallelization graph is also optimal.

Figure 5-8. Example of a merge-sort parallelizing template performing a lookup in a parallelization graph.

When a parallelizing template executes, it performs a lookup within the parallelization graph, based on the work metric of the input parameters, to return all of the information required to efficiently partition the computation and execute the recursive calls. For example, Figure 5-8 illustrates a merge-sort parallelizing template performing a lookup of 10,000, corresponding to the work metric of the invocation’s input parameters. The lookup returns the work metric, resources, and implementation to use for each of the recursive calls, which corresponds to the optimal selection for minimizing the maximum execution time as estimated by the function performance graphs. The parallelizing template then uses this information to partition the
computation accordingly and invoke the recursive calls using the corresponding resources and implementations.

5.3.5 Limitations

There are three main limitations of RACECAR. The first limitation is that the heuristic's implementation decisions will be less efficient if the implementation performance graph is less accurate. As described in Section 4.2.4, creating an implementation performance graph requires determining a relationship between the work metric and execution time. If an accurate mapping of the function's input parameters to work metric does not exist, or if the relationship between work metric and execution time have a large variance, then the resulting implementation performance graph will not accurately reflect the execution time of the implementation for different input parameters, and likewise the heuristic will make less efficient execution decisions. Fortunately, even for cases when the implementation performance graph is less accurate, the heuristic will still operate correctly and output execution decisions that work, albeit with reduced performance. In most cases, even the reduction in performance is negligible as the heuristic uses the implementation performance graphs to primarily select between implementations. Therefore, as long as the error is not significant enough to force the heuristic to select the wrong implementation, then the error may not even change the result. RACECAR also uses the implementation performance graphs to decide how to parallelize computation. An error in the implementation performance graph will influence how the heuristic decides to partition computation, but typically the errors are small when compared to the actual execution time, so the resulting performance loss is not significant. A more detailed discussion of
how to address the limitations for creating implementation performance graphs is presented in Section 4.2.4.

The second limitation of RACECAR is the required structure of parallelizing templates, where the template must first partition the computation, perform two recursive function calls, and then combine the results, as described in Section 5.2. Fortunately, many divide-and-conquer algorithms can be written to adhere to this structure. For those implementations that do not adhere to this structure, the parallelization graphs may still provide useful information. For example, the parallelizing template could perform a different lookup within the parallelization graph to compensate for the difference in usage, or just use the results of the parallelization graph as an approximation.

The third limitation of RACECAR is due to the assumptions made about the execution time of parallelizing templates. As described in Section 5.3.3, the heuristic assumes the most efficient way to partition computation for a parallelizing template is to minimize the maximum estimated execution time of the recursive calls. In reality, several more factors affect performance of the parallelizing template and could influence the best partitioning decision. For example, the partitioning and combining steps of the parallelizing template may vary with how the computation is partitioned, and therefore may also need to be taken into account. Additionally, the heuristic assumes the function performance graphs accurately reflect the performance of the recursive calls, but this does not include the inaccuracy of the implementation performance graphs or the interaction between the two functions when they execute simultaneously. Resource contention, caching affects, and system scheduling may all affect the
performance of the functions. Despite these limitations, the resulting error is typically small percentage-wise, and the resulting performance still yields significant speedups, as demonstrated in Chapter 6.

5.4 Elastic Function Execution

Elastic function execution occurs whenever an executing application (or a nested elastic function call in a parallelizing template) invokes the elastic function interface. For both cases, the elastic function call operates identically to a normal function call, with invoking code that first calls the interface with input parameters and then waits for the call to return any output parameters. Unlike a normal function call, the elastic function does not have a static implementation, so the call instead invokes the function execution tool to automatically select the best implementation for the current combination of input parameters, usage assumption, and availability of resources. The only difference between an elastic function invoked from an application and from a parallelizing template is in the specification of the resources on which an elastic function may execute. An application invoking an elastic function may utilize up to all of the resources available on a system. On the other hand, a parallelizing template is already executing on certain combination of resources, and therefore is limited to using only a subset of those resources for its internal nested elastic function calls.

Up until the point of elastic function execution, Elastic Computing has no knowledge of what input parameters the application will pass to the elastic function. Therefore, implementation assessment and optimization planning must have already predetermined how to efficiently execute the elastic functions using any input parameters, from which the function execution tool simply looks-up the execution decisions at run-time based on the actual input parameters. Delaying the determination
of how to execute the elastic function calls until run-time provides numerous benefits for the applications. First, Elastic Computing supports applications with dynamically changing input parameters. Second, the applications remain portable and adaptive to changes in how the elastic functions execute. For example, any changes in the hardware or improvements in how to execute the elastic functions can be instantly incorporated by all applications that use the elastic functions by simply re-performing implementation assessment and optimization planning, which will re-determine how to execute the elastic function efficiently using the new hardware or implementation improvements. At run-time, the function execution tool will automatically use the most up-to-date information for its execution decisions without requiring any recompilation of the applications.

The function execution tool operates as follows. First, the function execution tool uses the elastic function, usage assumption, and resources specified by the invocation to locate the corresponding FPG (created in Section 5.3.2). Second, the elastic function’s adapter converts the input parameters to the corresponding work metric. Third, the function execution tool uses the work metric to perform a lookup in the FPG, which returns the most efficient implementation and subset of resources to use for execution. Note that it is not always beneficial to use all of the available resources because more resources typically require more overhead, which the additional processing power will overcome only if the required computation is large enough. Fourth, the function execution tool allocates the subset of resources specified by the FPG from the set of available system resources, so that no other implementation may execute on the same resources before the elastic function completes. Fifth, the function
execution tool starts instances of the implementation specified by the FPG on all thread resources allocated to the implementation. This subset of resources may also include heterogeneous resources, but the function execution tool will only allocate the resources to the implementation and rely on the implementation’s code to actually interface with those resources, as described in Section 3.3.1. Lastly, the elastic function completes once all the thread instances terminate, at which point the function execution tool returns control to the invoking code. If the function execution tool selects a parallelizing template to execute, then the template will also perform lookups in the parallelization graph (created in Section 5.3.4) to determine how to partition computation between its nested elastic function calls.

![Diagram of function execution tool and parallelization](image)

**Figure 5-9.** An example of the function execution tool executing a sort for 10,000 elements on 4 threads/1 FPGA. Note that partition (#1) and combine (#1) are implemented within the code of parallelizing template (#1), and partition (#2) and combine (#2) are implemented within the code of parallelizing template (#2).

Figure 5-9 demonstrates the execution of a sorting elastic function on a system with four threads (i.e., cores) and an FPGA. Implementations contained within other implementations illustrate a parallelizing template invoking nested elastic functions. In
the figure, the application invokes a sort with an input of 10,000 elements. The function execution tool determines (based on the FPG) to implement the sort using a parallelizing template executing on three threads and an FPGA. The parallelizing template refers to the parallelization graph for the current execution situation, which specifies that the implementation should have one of its subsorts sort 4,000 elements, using the quick-sort implementation executing on a single thread, and have the second sub-sort sort 6,000 elements, using another parallelizing template executing on two threads and the FPGA. The second parallelizing template operates similarly and determines to parallelize its work with the insertion-sort implementation, executing on one thread, executing in parallel with a bitonic-sort implementation, executing on one thread and the FPGA. Once the insertion-sort and bitonic-sort complete, the second parallelizing template combines the results. Once the quick-sort implementation and the second parallelizing template complete, the first parallelizing template combines the results, which is returned to the application and completes the elastic function.

5.5 Summary

In this chapter, we described how Elastic Computing parallelizes, optimizes, and executes elastic functions. The optimizing step, called optimization planning, analyzes the estimated execution time of implementations, provided by the IPGs from Chapter 4, and relies on an algorithm, called RACECAR, that determines execution decisions to minimize the estimated execution time. The optimization planning decisions specify what implementation and resources to use to efficiently execute an elastic function, and additionally specify how a parallelizing template should partition its computation and resources between nested elastic function calls. Once optimization planning is complete, Elastic Computing saves the decisions for the function execution tool to refer
to at run-time when determining how to efficiently execute an elastic function on behalf of an application.
CHAPTER 6
EXPERIMENTAL RESULTS

In this chapter, we present experimental results assessing the ability for Elastic Computing to optimize elastic functions. Section 6.1 describes the hardware and software setup used to collect these results. Section 6.2 then analyzes the ability of Elastic Computing to accurately abstract and create performance predictors for implementations. Lastly, Section 6.3 looks at the overall speedup achieved by Elastic Computing’s optimization decisions.

6.1 Experimental Setup

To assess Elastic Computing, we selected ten functions and created a total of thirty-seven alternate heterogeneous implementations of those functions. Ideally, such implementations would be provided as part of a standard function library as described in Section 3.2.

We evaluated Elastic Computing on four diverse systems. The first system, named Delta, consists of a hyper-threading 3.2 GHz Intel Xeon CPU, 2 GB of RAM, and a Xilinx Virtex IV LX100 FPGA located on a Nallatech H101-PCIXM board. Hyper-threading makes Delta appear as though it has two cores, but the cores must partially contend for the same processing resources. The second system, named Elastic, consists of a 2.8 GHz quad-core Intel Xeon W3520 CPU, 12 GB of RAM, an Altera Stratix-III L340 FPGA located on a GiDEL PROCe III board, and two dual-chip Nvidia GTX-295 graphics cards (totaling four GPUs). The third system, named Marvel, consists of eight 2.4GHz dual-core AMD Opteron 880 CPUs (16 cores total) and 32 GB of RAM. The fourth system, named Novo-G, is a node of the Novo-G supercomputer [18] and
consists of a quad-core Intel Xeon E5520, 6 GB of RAM, and four Altera Stratix-III E260 FPGA’s located on a GiDEL PROCStar III board.

The Elastic Computing framework and CPU-based implementations were written in C++ and compiled using g++ with –O3 optimizations. The FPGA-based implementations were written in VHDL and compiled using Altera Quartus II version 9.1 (when targeting Elastic’s and Novo-G’s FPGAs) or Xilinx ISE 9.1 (when targeting Delta’s FPGA). Lastly, the GPU-based implementations were written in CUDA using compute capability 1.3 and compiled using nvcc release 4.0.

The thirty-seven implementations implement ten different functions. The ten functions come from a variety of problem domains and consist of: 1D convolution, 2D convolution, circular convolution, inner product, matrix multiply, mean filter, optical flow, Prewitt, sum-of-absolute differences image retrieval (SAD), and sort. The convolution functions perform one-dimensional or two-dimensional discrete convolution on two input operands. Inner product calculates the inner product of two input arrays. Matrix multiply multiplies two input matrices of compatible dimensions. Mean filter applies an averaging filter to an input image. Optical flow processes an input image to locate a feature. Prewitt performs Prewitt edge detection on an input image. SAD performs the sum-of-absolute differences image processing algorithm on an input image. Lastly, sort sorts an input array.

6.2 Implementation Assessment Results

We assess the ability of Elastic Computing to accurately abstract and create performance predictors for implementations by providing thirty-three implementations as inputs to the IA Heuristic and measuring the IPG creation time and estimation accuracy. To differentiate the implementations, we use the naming convention of starting with the
name of the function (except for sort which instead specifies the algorithm) and adding a
designation specifying the execution device. All non-heterogeneous implementations
are single-threaded unless they end with (MT) which specifies they are multi-threaded.
The heterogeneous implementations specify the particular heterogeneous device on
which they execute.

Table 6-1. Descriptions of the work metric mappings used for each function.

<table>
<thead>
<tr>
<th>Function</th>
<th>Work Metric Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D Convolution</td>
<td>Work metric equals product of dimensions of output matrix and dimensions of second input matrix (i.e., the sliding window).</td>
</tr>
<tr>
<td>2D Convolution</td>
<td>Work metric equals product of lengths of two input arrays.</td>
</tr>
<tr>
<td>Circular Convolution</td>
<td>Work metric equals product of lengths of two input arrays.</td>
</tr>
<tr>
<td>Inner Product</td>
<td>Work metric equals length of either input array.</td>
</tr>
<tr>
<td>Matrix Multiply</td>
<td>Work metric equals product of dimensions of first matrix and number of columns of second matrix.</td>
</tr>
<tr>
<td>Mean Filter</td>
<td>Work metric equals product of dimensions of output matrix and dimensions of filter.</td>
</tr>
<tr>
<td>Optical Flow</td>
<td>Work metric equals product of dimensions of output matrix and dimensions of second input matrix (i.e., the sliding window).</td>
</tr>
<tr>
<td>Prewitt</td>
<td>Work metric equals product dimensions of output matrix.</td>
</tr>
<tr>
<td>Sort (heap sort, insertion sort, quick sort)</td>
<td>Work metric equals length of input array.</td>
</tr>
<tr>
<td>Sum-of-Absolute Differences (SAD)</td>
<td>Work metric equals product of dimensions of output matrix and dimensions of second input matrix (i.e., the sliding window).</td>
</tr>
</tbody>
</table>

Table 6-1 describes the work metric mapping used by the adapters for each of the
ten functions. As described in Section 4.2, the implementation assessment step of
Elastic Computing abstracts the input parameter space of the implementation to allow the framework to analyze many types of functions. The work metric mapping is a critical part of this abstraction and significantly influences the accuracy of the resulting implementation performance graph.
Table 6-2. Work metric range and IPG creation time for the non-heterogeneous implementations.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Work Metric Range</th>
<th>Delta</th>
<th>Elastic</th>
<th>Marvel</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D Convolution</td>
<td>[0, 4.864e8]</td>
<td>29.2</td>
<td>23.1</td>
<td>28.6</td>
</tr>
<tr>
<td>2D Convolution (MT)</td>
<td>[0, 4.864e8]</td>
<td>40.4</td>
<td>7.1</td>
<td>2.4</td>
</tr>
<tr>
<td>Circular Convolution</td>
<td>[0, 1.536e9]</td>
<td>129.6</td>
<td>61.2</td>
<td>55.2</td>
</tr>
<tr>
<td>Circular Convolution (MT)</td>
<td>[0, 1.536e9]</td>
<td>100.1</td>
<td>12.2</td>
<td>4.1</td>
</tr>
<tr>
<td>Convolution</td>
<td>[0, 2.048e9]</td>
<td>100.3</td>
<td>48.3</td>
<td>74.7</td>
</tr>
<tr>
<td>Convolution (MT)</td>
<td>[0, 2.048e9]</td>
<td>112.6</td>
<td>12.7</td>
<td>5.4</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>[0, 5.000e6]</td>
<td>69.0</td>
<td>25.2</td>
<td>62.6</td>
</tr>
<tr>
<td>Inner Product</td>
<td>[0, 1.000e7]</td>
<td>18.7</td>
<td>5.4</td>
<td>10.4</td>
</tr>
<tr>
<td>Inner Product (MT)</td>
<td>[0, 1.000e7]</td>
<td>39.0</td>
<td>7.8</td>
<td>9.3</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>[0, 6.500e4]</td>
<td>63.1</td>
<td>33.8</td>
<td>42.8</td>
</tr>
<tr>
<td>Matrix Multiply</td>
<td>[1, 1.638e9]</td>
<td>83.9</td>
<td>42.9</td>
<td>65.1</td>
</tr>
<tr>
<td>Matrix Multiply (MT)</td>
<td>[1, 1.638e9]</td>
<td>78.5</td>
<td>12.6</td>
<td>7.1</td>
</tr>
<tr>
<td>Mean Filter</td>
<td>[0, 1.600e9]</td>
<td>105.3</td>
<td>45.9</td>
<td>97.7</td>
</tr>
<tr>
<td>Mean Filter (MT)</td>
<td>[0, 1.600e9]</td>
<td>107.2</td>
<td>9.6</td>
<td>9.0</td>
</tr>
<tr>
<td>Optical Flow</td>
<td>[0, 9.366e8]</td>
<td>74.0</td>
<td>38.8</td>
<td>45.9</td>
</tr>
<tr>
<td>Optical Flow (MT)</td>
<td>[0, 3.746e9]</td>
<td>361.1</td>
<td>43.0</td>
<td>13.4</td>
</tr>
<tr>
<td>Prewitt</td>
<td>[0, 1.000e8]</td>
<td>322.5</td>
<td>107.0</td>
<td>223.6</td>
</tr>
<tr>
<td>Prewitt (MT)</td>
<td>[0, 1.000e8]</td>
<td>504.9</td>
<td>114.7</td>
<td>123.8</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>[0, 1.000e7]</td>
<td>59.3</td>
<td>25.4</td>
<td>27.8</td>
</tr>
<tr>
<td>SAD</td>
<td>[0, 9.366e8]</td>
<td>71.3</td>
<td>38.2</td>
<td>49.4</td>
</tr>
<tr>
<td>SAD (MT)</td>
<td>[0, 3.746e9]</td>
<td>344.7</td>
<td>64.7</td>
<td>13.0</td>
</tr>
</tbody>
</table>
Table 6-3. Work metric range and IPG creation time for the heterogeneous implementations.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Work Metric Range</th>
<th>IPG Creation Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Delta</td>
</tr>
<tr>
<td>2D Convolution (Elastic GPU)</td>
<td>[0, 4.864e8]</td>
<td></td>
</tr>
<tr>
<td>Circular Convolution (Delta FPGA)</td>
<td>[0, 1.536e9]</td>
<td>35.1</td>
</tr>
<tr>
<td>Circular Convolution (Elastic GPU)</td>
<td>[0, 1.536e9]</td>
<td>2.1</td>
</tr>
<tr>
<td>Convolution (Delta FPGA)</td>
<td>[0, 1.843e9]</td>
<td>41.4</td>
</tr>
<tr>
<td>Convolution (Elastic FPGA)</td>
<td>[0, 2.048e9]</td>
<td>1.2</td>
</tr>
<tr>
<td>Convolution (Elastic GPU)</td>
<td>[0, 2.048e9]</td>
<td>1.7</td>
</tr>
<tr>
<td>Inner Product (Delta FPGA)</td>
<td>[0, 1.049e6]</td>
<td>2.4</td>
</tr>
<tr>
<td>Matrix Multiply (Delta FPGA)</td>
<td>[1, 1.638e9]</td>
<td>134.2</td>
</tr>
<tr>
<td>Matrix Multiply (Elastic GPU)</td>
<td>[1, 1.638e9]</td>
<td>24.4</td>
</tr>
<tr>
<td>Mean Filter (Elastic GPU)</td>
<td>[0, 1.600e9]</td>
<td>4.0</td>
</tr>
<tr>
<td>Optical Flow (Elastic GPU)</td>
<td>[0, 3.746e9]</td>
<td>12.0</td>
</tr>
<tr>
<td>SAD (Elastic GPU)</td>
<td>[0, 3.746e9]</td>
<td>13.5</td>
</tr>
</tbody>
</table>

Tables 6-2 and 6-3 list the work metric range (described in Section 4.2.3) and IPG creation time for each implementation. Table 6-2 lists only the CPU implementations while Table 6-3 lists the heterogeneous implementations. Note that the CPU implementations were executable on every system but the heterogeneous implementations only execute on specific systems. We determined the work metric range such that each implementation took around 3 seconds to complete when executing on a single-thread at the largest work metric value. The IPG creation time is measured as the total time required for the heuristic to analyze the implementation and create the IPG, including the time spent repeatedly executing the implementation during sample collection (Section 4.3.2).
As illustrated in the tables, the IPG creation time was quick for most implementations and systems. On average, the IA Heuristic required 65 seconds to create an IPG despite the implementations taking a few seconds to execute at larger work metric values. The maximum IPG creation time was less-than 10 minutes and 48 out of the 75 IPGs (64%) required less than 1 minute for the heuristic to create. The fastest IPG creation time required only 1.2 seconds.

The large variance in the IPG creation time is primarily due to the differing speeds of the systems. A faster system requires less time to execute the implementation, even for the same invocation parameters. The Elastic system, which is the fastest system, required an average of only 29 seconds to create an IPG. The slowest system, Delta, averaged around 121 seconds. The Marvel system was in the middle and averaged 46 seconds.
Figure 6-1. The estimation error of the IPG created by the IA Heuristic for 250 random invocations of each non-heterogeneous implementation. The bars specify the average estimation error and the lines specify the root-mean-squared error.
Figure 6-2. The estimation error of the IPG created by the IA Heuristic for 250 random invocations of each heterogeneous implementation. The bars specify the average estimation error and the lines specify the root-mean-squared error.

Figures 6-1 and 6-2 illustrate the average percentage of estimation error for each implementation’s IPG. We calculated the estimation error by averaging the percent difference between the measured and estimated execution time for 250 invocations of the implementations with random invocation parameters, generated by a Gibbs sampler. For all random invocations, we stress all degrees of freedom of the invocation parameters, such as using arrays with random lengths, matrices with random dimensions, and initializing all data structures with random values.

As illustrated in Figure 6-1, the IPG estimation error for the non-heterogeneous implementations varied widely between the different implementations and systems. 12 out of the 21 implementations (57%) had an IPG with an error of less-than 5% on at least one system, and only the matrix multiply implementations had IPGs with error larger than 35%. The largest error percentage for any IPG was 65%. In most cases, the
single-threaded versions of implementations produced an IPG of lesser error than their multi-threaded counterparts, due to the increased variability when executing on multiple cores.

The IPG estimation error for the non-heterogeneous implementations was strongly dependent on the executing system. The fastest system, Elastic, achieved an average estimation error of only 8%, while the slower systems of Delta and Marvel had average errors of 16% and 17% respectively. This discrepancy between the systems is due to the older systems having a larger penalty for a cache miss, resulting in a larger variance of execution time.

As illustrated in Figure 6-2, the IPG estimation error for the heterogeneous implementations varied even more significantly than the non-heterogeneous implementations. The IPG with the least error was for the Inner Product (Delta FPGA) implementation, which achieved an error of only 2%. The IPG with the largest error was for the Optical Flow (Elastic GPU) implementation having an error of 171%. 6 out of the 12 IPGs achieved an error of less-than 25% and only three IPGs had an error of over 100%. The variance varied significantly even in different implementations of the same function with the three convolution IPGs having errors of 18%, 75%, and 133%.

The large IPG estimation error is due to two main reasons. The first reason, which is especially evident for the non-heterogeneous implementations, is that some of the implementations have larger invocation-parameter spaces, which increases the difficulty of the work metric mapping to remain consistent over all possible invocation parameters. As illustrated in Figure 6-1, the implementations that accept two matrices for their inputs, (i.e., 2D Convolution, Matrix Multiply, Optical Flow, and SAD) all tended
to exhibit larger errors than those that only accepted two input arrays (i.e., Circular Convolution, Convolution, and Inner Product). The larger invocation-parameter space results in the work metric mapping needing to coalesce more invocation parameters to the same work metric value, thereby resulting in a larger variance for the execution time.

The second reason for the large IPG estimation error is that heterogeneous devices have numerous design and system factors that can significantly influence their execution time. Design factors include how efficient the heterogeneous implementation executes over the full range of invocation parameters. For example, some GPU implementations, which internally rely on partitioning computation over possibly hundreds of light-weight threads, may not execute as efficiently unless certain dimensions of the input parameters are multiples of the number of threads (e.g., the image for a 2D convolution implementation having a width that is a multiple of the number of threads). System factors include the overhead of communicating and synchronizing with the heterogeneous device, which at times can require more time than the computation itself. As listed in Table 6-1, we based the work metric mappings of all the implementation adapters on the asymptotic analysis technique discussed in Section 4.2.2, which assumes the execution time of an implementation is largely dependent on the amount of computation it requires. However, when other factors affect the execution time, this assumption becomes less accurate resulting in a larger variance.

6.3 Elastic Function Speedup Results

We assess the ability of Elastic Computing to make efficient execution decisions by measuring the performance gain Elastic Computing achieves for ten elastic functions
executing on three heterogeneous systems and constructed with a total of thirty-seven heterogeneous implementations. As discussed in Chapter 5, the thirty-seven implementations provide the implementation alternatives for the function’s computation from which Elastic Computing will automatically determine efficient implementation selection and parallelization decisions.

The thirty-seven implementations consist of microprocessor and heterogeneous implementations. All ten functions contain a single-threaded microprocessor implementation except for sort, which contains three microprocessor implementations corresponding to different sorting algorithms (insertion sort, heap sort, quick sort). We created Elastic FPGA implementations for the 1D convolution, 2D convolution, optical flow, and SAD functions. We created Elastic GPU implementations for the 1D convolution, 2D convolution, circular convolution, matrix multiply, mean filter, optical flow, and SAD functions. We created Novo-G FPGA implementations for the 1D convolution, 2D convolution, optical flow, and SAD functions. While ideally all functions would have heterogeneous implementations, time constraints required us to create implementations for only those functions we predicted significant speedup. Lastly, all ten functions also contain a parallelizing implementation.
Figure 6-3. The speedup achieved by Elastic Computing for each elastic function. All speedup numbers are relative to a single-threaded implementation of the function on the corresponding system. The shorter bar reflects the speedup achieved by the fastest single implementation alone while the longer bar is the speedup achieved by Elastic Computing.

Figure 6-3 illustrates the Elastic Function speedup of each function on each system. All speedup numbers are relative to a single-threaded implementation of that function executing on the same system. To provide a fair comparison for what a developer might normally create to take advantage of a heterogeneous system (e.g., a single efficient FPGA or GPU implementation), the figure also shows the speedup achieved by the fastest single implementation provided as an input for that function. Elastic Computing achieves a speedup faster than its implementation inputs by parallelizing computation across multiple resources.
The speedup of each function was highly dependent on the implementations supported by the executing system. For functions executing with only microprocessor implementations, RACECAR was typically able to approach linear speedup with the number of microprocessors. For example, the Marvel system, which contains 16 microprocessors, was able to achieve greater than 14x speedup on 8 out of the 10 functions. For functions that contained a single very fast heterogeneous implementation (e.g., the functions with Novo-G FPGA implementations), the speedup was largely dictated by the single implementation. This result is due to the heterogeneous implementation dwarfing any speedup possible by parallelizing the computation across multiple, slower resources.

The Elastic system provided the most interesting results as it contains multiple fast heterogeneous resources. The fast GPU and FPGA implementations provided as inputs to Elastic Computing were able to, by themselves, achieve speedup averaging 45x when compared to a microprocessor implementation. Despite the already fast implementations, Elastic Computing was able to further increase speedup by having the GPU and FPGA implementations execute in parallel. For the fastest implementation of 1D convolution, which uses the FPGA, Elastic Computing was able to increase speedup from 55x to 86x by determining how to parallelize computation across both the GPUs and FPGAs. Similarly, mean filter was able to increase speedup from 25x to 44x by taking advantage of the additional resources. Nearly all the functions on Elastic follow this trend. Prior to Elastic Computing, determining how to effectively utilize multiple heterogeneous resources in a system was a laborious manual process. By using Elastic
Computing, the optimization steps automatically incorporate multiple fast individual implementations to make them even faster.

Figure 6-4. The speedup achieved by Elastic Computing averaged over each system. All speedup numbers are relative to single-threaded implementations of the functions on the corresponding system. The shorter bar reflects the speedup achieved by the fastest single implementations alone while the longer bar is the speedup achieved by Elastic Computing.

Figure 6-4 illustrates the average speedup achieved on each system by both the fastest single implementation and Elastic Computing. On all systems, Elastic Computing was able to effectively utilize the parallel resources to significantly improve performance beyond any single implementation by itself. For both Elastic and Novo-G, the faster heterogeneous resources already provided a large speedup over the microprocessor implementations. None the less, Elastic Computing was still able to average a 1.3x improvement over the single implementations. For Marvel, all of the implementations were microprocessor implementations, allowing Elastic Computing to further increase performance by an average of 12x by parallelizing computation over the numerous CPUs. Overall, Elastic Computing was able to achieve an average function speedup of 47x, while the fastest single implementations were only able to average 33x.
CHAPTER 7
CONCLUSIONS

In this document, we presented an optimization framework, called Elastic Computing, that is capable of separating functionality from implementation details, allowing application designers to more easily exploit the performance potential of multi-core heterogeneous systems. With Elastic Computing, the application designer simply specifies functionality in terms of elastic functions, which the Elastic Computing tools convert into specialized implementations through a combination of implementation assessment, optimization planning, and elastic function execution. We evaluated Elastic Computing on four diverse systems, showing that the framework invisibly achieved speedup (no coding changes were required) for different resource amounts. Furthermore, we showed that Elastic Computing can adapt to the run-time conditions of the functions, such as the function’s input parameters, achieving performance significantly better than the individual implementations, even with the overhead of elastic function execution. Lastly, we demonstrated the significant performance improvement Elastic Computing achieves on a wide variety of functions and multi-core heterogeneous systems.

Although Elastic Computing focuses primarily on multi-core heterogeneous systems, the enabling technologies have wide-spread applicability. The IA Heuristic used during implementation assessment (Chapter 4) enables very flexible performance prediction, which is an essential task in most kinds of optimization, such as with compilers or design tools. The heuristic analyzes an implementation only in terms of the execution time required to complete for different input parameters, effectively treating the implementation as a black box. As a result, the IA Heuristic can analyze many
different types of implementations, even those written in different programming languages or executing on different devices.

The RACECAR Heuristic used by optimization planning (Chapter 5) is similarly very flexible and can make efficient execution decisions with only the information provided by performance predictors. RACECAR only analyzes the performance predictors directly, which could be created by the IA Heuristic or perhaps another method, allowing RACECAR to partition computation between many types of implementations including those written in different programming languages or executing on different devices. The performance predictors could even represent different nodes in a cluster, giving RACECAR applicability to high-performance cluster computing or even cloud computing.

The combination of the IA and RACECAR heuristics make Elastic Computing both flexible and powerful. Measured results presented in Chapter 6 demonstrate Elastic Computing results for ten elastic functions executing on four diverse heterogeneous systems. On the systems with multiple CPUs, Elastic Computing was able to achieve nearly linear speedup with each additional core. On the heterogeneous systems, the performance benefit was dominated by the provided heterogeneous implementations, but none the less Elastic Computing was still able to further optimize computation and achieve a speedup of up to 233x when compared to a single-threaded execution.

Developing applications to execute efficiently on multi-core heterogeneous systems is one of the largest challenges in software engineering today. While the hardware continues to evolve and improve in performance capability, the software engineers have largely been playing catch-up due to trying to learn a multitude of tools
and programming languages to take full advantage of newer hardware. Very few practical tools exist to perform full-system heterogeneous optimization. Additionally, most of these tools and newer technologies are aimed at experts, which limit the average developers with marginal performance improvements on new systems. Elastic Computing provides a significant step towards solving these problems and enabling effective multi-core heterogeneous computing for all developers.
REFERENCES


BIOGRAPHICAL SKETCH

John Robert Wernsing is currently a software engineer at Google, Inc. and lives in Seattle, WA. John is also a Ph.D. graduate from the Department of Electrical and Computer Engineering at the University of Florida. John received his M.S. degree from the Department of Computer and Information Science and Engineering at the University of Florida in May of 2011. For his undergraduate studies, John received his B.S. degree in Computer Engineering, graduating Summa Cum Laude, and B.S. degree in Electrical Engineering, graduating Cum Laude, from the University of Florida in May of 2006, where he also was a recipient of the Department of Electrical and Computer Engineering’s “Electric E” award. During his undergraduate studies, John also interned at Advanced Micro Devices, Microsoft, and Motorola.