Elastic Computing: A Framework for Transparent, Portable, and Adaptive Multi-core Heterogeneous Computing

John R. Wernsing, Dr. Greg Stitt

University of Florida
Department of Electrical & Computer Engineering
Gainesville, FL, USA
wernsing@ufl.edu, gstitt@ece.ufl.edu

Abstract

Over the past decade, system architectures have started on a clear trend towards increased parallelism and heterogeneity, often resulting in speedups of 10x to 100x. Despite numerous compiler and high-level synthesis studies, usage of such systems has largely been limited to device experts, due to significantly increased application design complexity. To reduce application design complexity, we introduce elastic computing – a framework that separates functionality from implementation details by enabling designers to use specialized functions, called elastic functions, which enable an optimization framework to explore thousands of possible implementations, even ones using different algorithms. Elastic functions allow designers to execute the *same application code* efficiently on potentially any architecture and for different runtime parameters such as input size, battery life, etc. In this paper, we present an initial elastic computing framework that transparently optimizes application code onto diverse systems, achieving significant speedups ranging from 1.3x to 46x on a hyper-threaded Xeon system with an FPGA accelerator, a 16-CPU Opteron system, and a quad-core Xeon system.

Categories and Subject Descriptors J.6 [*Computer-Aided Engineering*]: Computer-aided design (CAD).

General Terms Performance, Design

Keywords elastic computing; heterogeneous architectures; multi-core; FPGA; speedup

1. Introduction

The power bottleneck caused by increasing clock frequencies has led to a trend towards increased parallelism, most notably with multi-core microprocessors [13][15][28], in addition to increased diversity via heterogeneous processing resources specialized for different tasks. Such heterogeneity often includes accelerators such as field-programmable gate arrays (FPGAs) and graphics processing units (GPUs), which numerous studies have shown to achieve 10x to 100x speedups over microprocessors for many applications [5][11][18]. Due to these significant performance improvements, the combination of multi-cores and heterogeneity, referred to as *multi-core heterogeneous systems* for simplicity, is becoming increasingly common in domains ranging from low-power embedded systems [6][28][32], to high-performance embedded computing [16][31], to high-performance computing (HPC) systems [4][24].

Although multi-core heterogeneous systems provide significant improvements, effective use of such systems has mainly been limited to experts due to an increase in application design complexity. Numerous approaches have aimed to reduce design complexity for such systems by hiding low-level details using improved compilation [9] and high-level synthesis [12][26]. Similarly, new languages have been introduced to ease parallel programming [2][3][7].

Although these previous approaches have had some impact on productivity, a fundamental limitation of previous work is the specification of an application as a single implementation. Much prior work [10][27] has shown that different implementations of the same application often have widely varying performances on different architectures, which we refer to as the *implementation portability problem*. For example, a designer implementing a sorting function may use a merge-sort or bitonic-sort algorithm to create an FPGA implementation but a quick-sort algorithm to create a microprocessor implementation. Furthermore, this problem extends beyond efficiency for a particular architecture. Different algorithms operate more efficiently for different input sizes [23], different amounts of resources [8], and potentially any other runtime 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. 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 and algorithm 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 runtime conditions. Elastic computing, shown in Figure 1, is a framework, combining standard application code – potentially written in any language – with a library of specialized *elastic functions*, an *implementation planning* tool, and an *elastic computing system* runtime environment. As shown in Figure 1(a), elastic functions specify multiple implementations of the same functionality, possibly including calls to other elastic functions, which enables implementation planning to explore and even generate thousands of new implementations specialized for different situations such as

LCTES'10 April 13–15, 2010, Stockholm, Sweden.

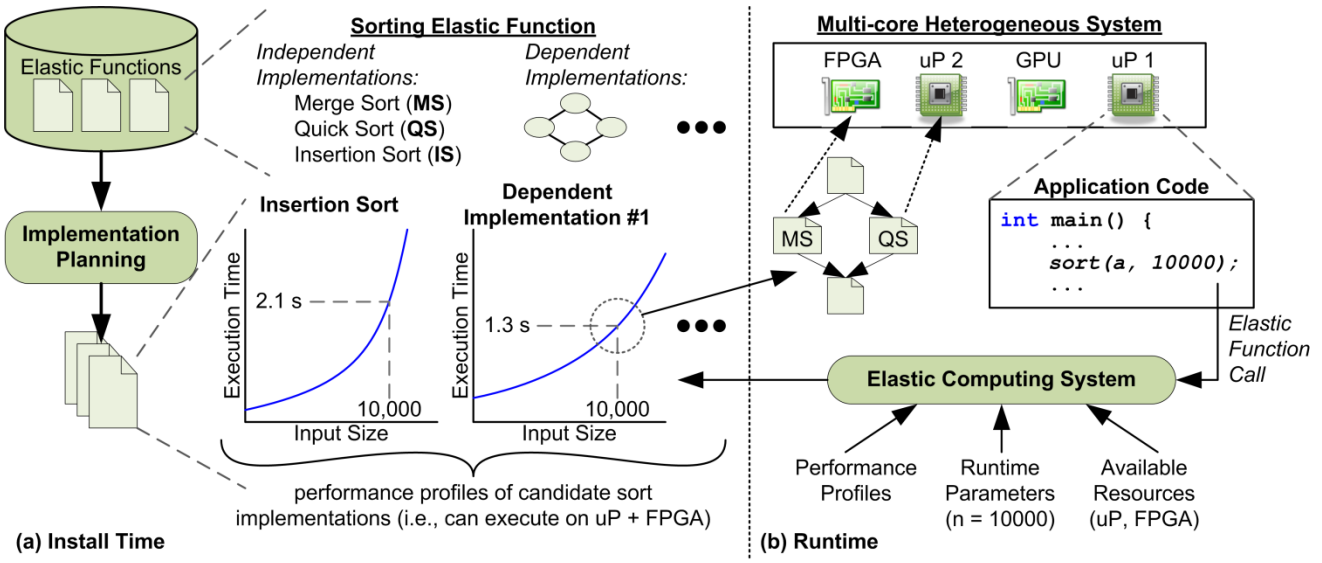

Figure 1. An overview of *elastic computing*, which is enabled by (a) *elastic functions* that enable *implementation planning* to explore and even generate different implementations specialized for parameters such as input size, available resources, etc. (b) When an executing application calls an elastic function, the *elastic computing system* selects the quickest implementation based on the current runtime parameters and available resources. Note that no changes to the application code are required to use different resources.

different input sizes, resource combinations, and potentially even remaining battery life, current power consumption, etc. Note that the naming convention used in this paper is to refer to an algorithm using hyphened form (e.g., merge-sort) and the corresponding implementation using proper noun form (e.g., Merge Sort).

One key advantage of elastic computing is the complete separation of functionality from implementation details. As shown in Figure 1(b) for a sorting example, the application designer simply calls an elastic sorting function without specifying how that sort is implemented. Instead, the elastic function call invokes the elastic computing system which analyzes runtime parameters and selects the most efficient implementation for the current situation by using performance profiles previously determined during implementation planning. Thus, without any effort or knowledge of the architecture, the application designer in this example is able to execute a sorting implementation that elastic computing automatically specializes for the current architecture, taking advantage of the FPGA as well as a microprocessor. While previous work has shown such optimization for specific systems, applications, and languages, to our knowledge, elastic computing is the first generalized technique that potentially enables invisible optimization of any application for any system.

Elastic computing is largely intended to enable mainstream application designers, who often lack the skills required for programming specialized devices, to take advantage of such devices with minimal effort. Of course, for elastic computing to be widely used, an elastic function library must be provided to these application designers. We envision that such a library could be created for different domains, with implementations being provided by device vendors (e.g., Xilinx, Nvidia) interested in attracting a new market of users, third parties (e.g., Rapidmind [20]), or even by open-source efforts. As opposed to only improving productivity of mainstream application designers, elastic computing also provides mechanisms that make implementation design of elastic functions less complex than existing multi-core heterogeneous design. A complete discussion of possible usage scenarios is discussed in Section 6.

The paper is organized as follows. Section 2 discusses related work. Section 3 defines elastic functions. Section 4 discusses implementation planning. Section 5 describes the elastic computing system. Section 6 summarizes elastic computing usage scenarios, advantages, and limitations. Section 7 presents experimental results.

2. Related Work

The implementation portability problem was addressed by Grattan [10], who introduced codesign-extended applications that specified multiple implementations of a function, which enabled a compiler to explore multiple possibilities for hardware and software implementations. Although that approach achieved improvements in portability and efficiency, application designers had to manually specify multiple 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 elastic functions, with efficient implementations of those functions determined by the elastic computing system.

Previous work on adaptable software also shares similarities with elastic computing. FFTW (Fastest Fourier Transform in the West) [8] 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) [29] is a similar library of automatically-tuned sparse matrix kernels. ATLAS [30] is a software package of linear algebra kernels that are capable of automatically tuning themselves to different architectures. Such approaches are essentially examples of manually created elastic functions for particular devices. PetaBricks [1] consists of a language and compiler that enables algorithmic choice, but restricts parallelizing decisions to static choices. Qilin [17] can dynamically determine an effective partitioning of work across heterogeneous resources, but targets data-parallel operations. Elastic computing aims to provide a general framework that enables any elastic

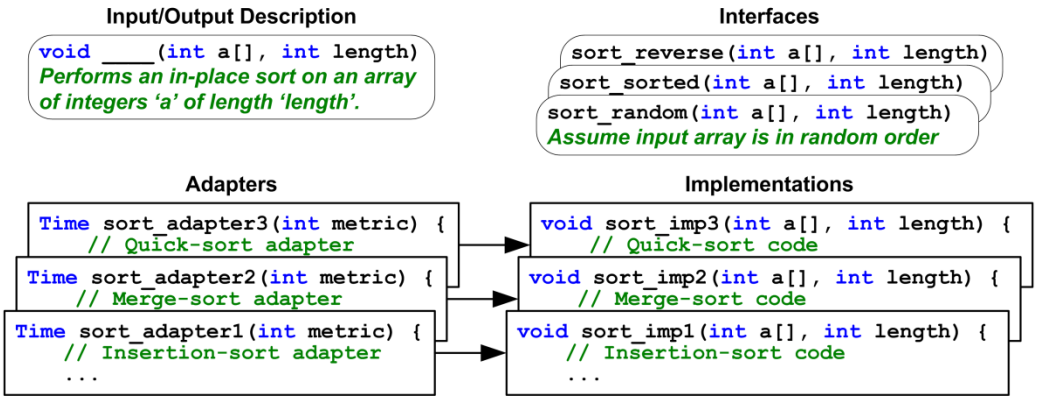

Figure 2. Components of an example sorting elastic function.

function 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 homogeneous architectures, elastic computing can potentially be used with any multi-core heterogeneous architecture and can also adapt to runtime changes.

3. Elastic Functions

Elastic functions form the basis of elastic computing by hiding implementation details, allowing the application designer to simply call the elastic function and rely on the elastic computing system to make all of the implementation decisions. As shown in Figure 2 for a sorting example, elastic functions consist of four components: an *input/output* description, a set of *interfaces*, a set of *implementations,* and an *adapter* for each implementation.

The input/output description defines the input and output parameters for the elastic function, similar to a C-function prototype. Although not used by the elastic computing system, the input/output description also includes a semantic description of the parameters. For example, in Figure 2, the input/output description specifies that the function accepts two parameters, the first parameter being a pointer to an integer array which should be sorted in-place, and the second parameter being the size of the array.

Elastic function interfaces are function prototypes that are exposed to standard programming languages, which enable an application designer to invoke an elastic function. Unlike standard functions, elastic functions provide one or more interfaces that enable the application designer to inform the elastic computing system of specific assumptions. For example, because different sorting implementations have different performances based on the characteristics of the input data (e.g., randomly distributed, mostly sorted), interfaces enable the designer to describe those characteristics so that the elastic computing system can make better implementation selection decisions. As shown in Figure 2, the elastic function can provide a separate interface for sorting randomly distributed data, in which case the elastic computing system may select a Quick Sort implementation, in addition to an interface for sorting mostly sorted data, in which case the elastic computing system may select an Insertion Sort implementation. Each interface is invoked identically (i.e., adheres to the same input/output description).

The main difference between standard functions and elastic functions is that elastic functions define one or more possible implementations, each of which adheres to the input/output description. We categorize the implementations into two groups: *independent implementations* and *dependent implementations*. Independent implementations are binary executables for a specific combination of resources, which can be created using any language or compiler/synthesis tool. For example, a sorting elastic function may have independent implementations that rely on different algorithms (e.g., quick-sort, insertion-sort, merge-sort) and different resource combinations (e.g., microprocessor, FPGA, GPU, microprocessor+FPGA, microprocessor+GPU).

Dependent implementations are defined similarly to independent implementations, but also internally call one or more elastic functions (i.e., the implementation *depends* on functionality provided by other elastic functions). For each elastic function call in the dependent implementation, the elastic computing system selects a corresponding implementation at runtime, which may be dependent or independent. For example, a dependent implementation of a *Sort* elastic function may internally rely on elastic functions for *Split*, *Merge*, and *Sort*. By referring to other elastic functions, dependent implementations effectively create degrees of freedom that enable the elastic computing system to create completely new implementations for different situations. Dependent implementations also have the advantage of specifying explicit task-level parallelism.

Each implementation additionally provides an adapter that allows implementation planning to configure the input parameters of the implementation, as described in detail in the following section.

4. Implementation Planning

Implementation planning analyzes elastic function implementations and the execution resources of the system to determine a "plan" that identifies the most efficient implementation of each elastic function for all combinations of resources, input parameters, and interfaces. Such planning is necessary to minimize the runtime overhead of implementation selection in the elastic computing system, which is discussed in Section 5. Implementation planning executes when the elastic computing framework is installed on a system, when the elastic function library changes (e.g., new implementations are added), or when new resources are added to a system (e.g., adding an FPGA board).

Implementation planning outputs a *performance profile* for every valid combination of elastic function interface, execution resources, and implementation. The performance profile consists of a set of statistically significant points, referred to as *significant points*, that represent execution times for different input parameters, which the elastic computing system can use to estimate the

```
ImplementationPlanning( e_library, resources )
begin
  do
    foreach Combination r of resources
      foreach Interface e in e_library
        foreach Implementation i that supports e
          a = i.GetAdapter();
          profiles[r,e,i] = a.CreateProfile(r,e);
        endfor;
      endfor;
    endfor;
  loop until AllProfilesStabilized(profiles);
  return profiles;
end;
```

Figure 3. High-level steps of implementation planning.

execution time of an implementation for any combination of input parameters via linear interpolation. Specifically, the performance profile represents a two-dimensional plot with an abstraction referred to as the *metric* on the X-axis and execution time on the Y-axis. Implementation planning internally uses the metric as an abstract representation of the input parameters to an implementation. The adapter for each implementation provides a mapping between metric and input parameters. For example, one possible adapter for an Insertion Sort implementation may map the metric value to the size of the sort. In this example, the estimated execution time for any invocation of the Insertion Sort can be found by simply looking up the Y-value (i.e., execution time) in the performance profile when the X-value (i.e., metric) is equal to the size of the sort. More complicated adapter mappings are discussed in Section 4.1.

By comparing the performance profiles of multiple implementations for an elastic function interface, the elastic computing system can determine the fastest overall implementation for a given combination of execution resources and input parameters. For example, Figure 1 demonstrates a sorting elastic function with several possible implementations (e.g., Insertion Sort, Merge Sort, Quick Sort). For simplicity, the example assumes that the adapter maps input size directly to metric. The call to the **`sort`**() from the application code specifies that a sort for 10,000 elements is required. The elastic computing system uses the performance profiles for each of the implementations, evaluated at a metric of 10,000, to estimate the execution time of each implementation. The elastic computing system then selects the implementation with the fastest estimated execution time, a dependent implementation in this case, and initiates its execution on the available execution resources. Note that this is just one example and different input parameters, implementations, or execution resources could affect the decision.

Figure 3 shows the high-level steps of the implementation planning algorithm. The input to the algorithm is a library of elastic functions (**`e_library`**) and the resources available on the system (**`resources`**). For each iteration of the innermost loop, the algorithm first obtains the implementation's adapter to convert the profiled metric values to the input parameters (Section 4.1), and then creates a new performance profile using that adapter for a specific combination of elastic function interface and execution resources, using a heuristic described in Section 4.2. For dependent implementations, the profiles of other implementations may affect the performance of the current implementation. As a result, the outermost loop repeats the profiling process until all the profiles stabilize, as described in Section 4.3. Section 4.4 discusses limitations of the algorithm.

4.1 Adapter

One challenge is that implementation planning must create performance profiles for any type of implementation and elastic function interface. Different implementations require different types and amounts of resources to execute, and every implementation has varying requirements on its input and output parameters. Some implementations may even require allocation and initialization of secondary data structures (e.g., a sorting implementation expecting an array of integers as input). Additionally, different elastic function interfaces may require different types of parameter initializations (e.g., populating an array with randomly distributed data as opposed to mostly sorted data). Abstracting these implementation and elastic function interface details are the purpose of the adapter.

The implementation planner relies solely on an abstract value, referred to as the metric, to represent the input parameters for any implementation. The adapter internally provides mappings between the metric and input parameters. The adapter also provides functionality to empirically measure the execution time of an implementation for a given metric value. Specifically, the implementation planner can call the adapter passing in a metric value. The adapter internally performs any necessary initialization to execute the implementation, maps the metric value to corresponding input parameters, and then executes the implementation while measuring its execution time. The adapter then returns the resulting execution time, which the heuristic can use to locate the significant points and construct the performance profile, as described in Section 4.2.

The complexity of the adapter itself depends mostly on the difficulty of creating a mapping between input parameters and the metric. Many implementations can have their execution time predominantly dictated by a single input parameter. For example, the size of the input array predominantly determines the execution time for an Insertion Sort implementation. In these cases, an adapter design would typically map the metric directly to that input parameter (e.g., map the metric to be the size of the sort) and have the remaining parameters appropriately populated (e.g., allocate an array of randomly distributed data to be sorted). For more complicated implementations, other techniques such as algorithmic complexity analysis can be used to determine a mapping. Algorithmic complexity analysis takes advantage of an in-depth understanding of the factors involved in the algorithm's execution time to find an appropriate metric mapping. For example, one algorithm to implement circular convolution results in a complexity analysis of $\Theta(|a|*|b|)$, that is the execution time is proportional to the product of the two input operands. An obvious direct mapping in this case is not possible as the size of both operands significantly affect the execution time of the implementation. However, if it is assumed that the proportionality factor in the complexity analysis remains approximately constant, then one possible adapter for this algorithm would be to map the metric to be the length of one of the operands and to fix the length of the second operand to be one. The reverse mapping from input parameters to metric would then be to set the metric equal to the product of the lengths of the two input parameters. This mapping works as the product is the same. For example, if the performance profile specifies that convolving 1,000,000 elements (mapped to metric) and 1 element (constant) takes 2 seconds, then the estimated execution time for convolving 1,000 elements and 1,000 elements should also be approximately 2 seconds.

To maximize prediction accuracy, the adapter should ideally meet specific assumptions made by the performance profile creation heuristic. First, the heuristic assumes that execution time is non-decreasing with increasing metric. Second, the heuristic assumes that execution time changes smoothly with small metric changes (i.e., no abrupt jumps). Lastly, the heuristic assumes that for a specific metric value, execution time is distributed normally amongst a constant mean. The closer the adapter meets these assumptions, the more accurate the resulting performance profile will be. However, even if the adapter does not perfectly meet any of these assumptions, the resulting profile is typically still usable albeit with reduced accuracy.

4.2 Performance Profile Creation

This section describes a heuristic for creating a performance profile for a single implementation on a given set of resources. Creating the performance profiles is one of the main challenges of implementation planning because unlike previous performance prediction work [14][25], each profile must predict performances for all combinations of input parameters.

The basic operation of the heuristic is to generate the performance profile by locating the metric value of each significant point based on prediction parameters discussed in the next paragraph. The heuristic estimates the location of significant points by empirically measuring the execution time of the implementation at several metric values, a process we refer to as *sampling*, and then analyzing sets of nearby samples using linear regression analysis.

The prediction parameters are defined when the elastic function library is installed, enabling different systems to tune the heuristic for specific devices. The first prediction parameter, which we refer to as α%, specifies the maximum allowed percent error of the performance profile. If it is assumed that execution time is non-decreasing with increasing metric, then spacing the significant points α% apart, as illustrated in Figure 4, limits the maximum percent error for any point in between the significant points to be α%. The second prediction parameter, which we refer to as β%, specifies the level of required confidence in the location of each significant point. Specifically, the parameter specifies the maximum percent width allowed for a confidence interval relative to its location. For example, if the linear regression analysis specifies that the execution time for a specific metric is 5±1 seconds, which is also equal to 5±20% seconds, the accuracy would not be deemed sufficient if β% was less-than 20%. The third prediction parameter is the confidence level used for confidence interval calculations. The last prediction parameter is the range of metric values for which the heuristic should generate the performance profile. The range of metric values may be different for every implementation.

Figure 4 illustrates the detailed operation of the heuristic. First, the heuristic starts at the smallest metric value, referred to as m_0, and collects enough samples at that metric value to be able to determine the execution time, referred to as t_0, for that metric within β% accuracy. The resulting (m_0, t_0) point is the first significant point of the performance profile. Second, the heuristic increments the execution time of the previous significant point (henceforth referred to a t_{n-1}) by α% to determine the execution time of the current significant point ($t_n = t_{n-1}+\alpha\%$). Third, the heuristic identifies a subset of the samples, out of all the samples it has taken thus far, on which to perform a linear regression analysis for determining if it already knows the metric value of the current significant point (i.e., what m_n would yield an execution time of t_n) within the required β% accuracy. One way to pick this subset would be to consider only samples with an execution time between t_{n-1} and t_{n+1} (a more robust approach is discussed in the following paragraph). Fourth, if there are not enough samples to determine the current significant point, the heuristic measures another sample and the process repeats. One way the heuristic could determine the metric value to sample would be to randomly pick a metric within the range between m_{n-1} and the metric of the first sample that has an execution time greater than t_{n+1} (a more robust approach is discussed in the next paragraph). Lastly, once the heuristic has enough samples to determine the current significant point within β% accuracy, the heuristic appends the significant point (m_n, t_n) to the performance profile and the process repeats with the next significant point. The heuristic completes once the linear regression analysis reveals that the current significant point would have a metric value greater-than the upper bound.

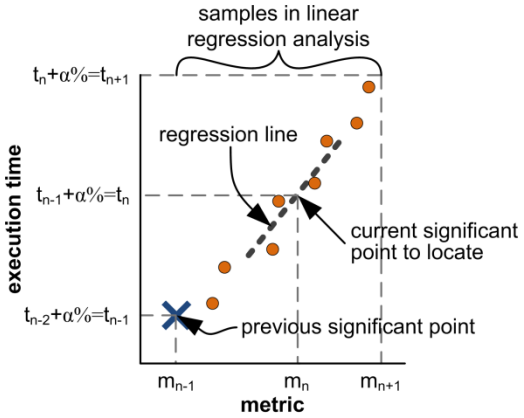

Figure 4. Illustration of performance profile generation.

The procedure of selecting samples for the linear regression analysis, used in steps three and four of the previous discussion, made the assumption that samples are non-decreasing with increasing metric, which is generally not true as there is often some variance in execution time. A more robust approach is to not filter out samples based on their execution time alone, but instead filter based on the regression of the samples. To perform this filtering, the heuristic first sorts all of the samples in ascending metric order, finds the first sample with a metric greater-than or equal to the previous significant point's metric (m_{n-1}), and inserts that sample into a set S. Second, the heuristic inserts the next two samples, in sorted increasing metric order, also into S. Three samples are initially required to avoid a non-trivial linear regression analysis. Third, the heuristic performs a linear regression analysis on the samples in S and calculates the execution time that the regression line has for the largest metric in S, which is also the last sample in S as the samples are sorted. Fourth, if this calculated execution time is less-than t_{n+1} then the heuristic adds the next sample to S, also in increasing metric order, and repeats. Lastly, if the calculated execution time is greater-than or equal to t_{n+1} then the process is complete. By filtering based on the regression, the susceptibility of the heuristic to the variance inherent in the execution time of individual samples is reduced and the entire process becomes more robust.

Although a complete evaluation of the performance profile creation heuristic is outside the scope of this paper, we summarize the results as follows. In a test involving twelve different implementations having a variety of time complexities, with prediction parameters set to α% = β% = 5%, the heuristic on average required only 374 samples and created a performance profile with an average estimation error of 5.84% (calculated using 250 random combinations of input parameters for each implementation).

In fact, five of the twelve implementations had a prediction error of less than 1%, with only one of the implementations having an error of greater than 12%. Additionally, the tests showed that when the prediction parameters were set to $\alpha\% = \beta\% = 50\%$ (a 10x increase in allowed error), the average number of samples dropped to 52, resulting in a 6x speedup of the heuristic, yet the average estimation error increased only to 6.1%.

4.3 Performance Profile Stabilization

Because dependent implementations internally call elastic functions, their performance profiles are dependent on the performance profiles of the fastest implementation of each called elastic function. This interdependence results in performance profiles that may improve (i.e., are unstable) after every iteration of the implementation planning algorithm. For example, if an updated performance profile reveals a quicker implementation for sorting 10,000 elements, this would improve the execution time and change the resulting performance profile for any dependent implementation of any elastic function that happens to internally call a sort for 10,000 elements.

Implementation planning iterates until the performance profiles show no improvement, which we refer to as profile stabilization. An upper bound on the number of iterations required for stabilization is equal to the deepest call-stack achievable by the dependent implementations (regardless of which elastic functions are actually called). In practice, the profiles often stabilize after only a few iterations due to dependent implementations being used only to parallelize the elastic function, and independent implementations being used to actually perform the computation on individual resources. As a result, the dependent implementation call stack is only a few levels deep reflecting only the partitioning required by the elastic function. For the evaluated applications, the profiles stabilized after only three iterations.

4.4 Limitations

The main limitations of the implementation planning algorithm correspond to difficulties in creating an effective adapter for an implementation. First, an implementation must provide a way to map a metric to input parameters. Implementations that are not deterministic or exhibit widely varying execution times are not suitable. Second, architecture specific effects may reduce the accuracy of the performance profiles. Cache flushing, data alignment, and CPU pre-emption will add a level of variation to execution time measurements. However, in most cases these variations are relatively small and can be ignored. Third, some performance profiles will have reduced accuracy for corner cases. Corner cases are small subsets of input parameter combinations to an implementation that force the execution time to not follow the general trend. For example, a Quick Sort implementation exhibits an algorithmic time complexity of $O(n \log n)$ for randomly distributed data but $O(n^2)$ for already sorted data, where 'n' is the number of elements to sort. In this case, a performance profile generated using randomly distributed data would have a large percent error if used to estimate the execution time for sorting already sorted data. However, elastic computing alleviates the affect of corner cases by enabling the designer to specify usage assumptions, such as by having separate interfaces for sorting randomly distributed data and sorting mostly sorted data.

5. Elastic Computing System

The elastic computing system, illustrated in Figure 5, serves two main purposes. First, when an application invokes an elastic function, the elastic computing system analyzes the current available resources and invocation parameters of the elastic function, and then selects the fastest implementation based on the performance profiles from implementation planning. Second, the elastic computing system provides runtime services (e.g., resource allocation, device abstraction, and communication), to support the implementations until the elastic function is complete.

The elastic computing system is invoked whenever an application or a dependent implementation calls an elastic function via one the function's interfaces. When an application calls an elastic function, the elastic computing system will by default allocate all available resources to that elastic function, although the application designer has the option to explicitly state which resources to use. When a dependent implementation calls an elastic function, the implementation itself allocates a subset of its resources to use for that call.

With the combination of the elastic function invocation parameters, the list of resources, and the corresponding interface used by the application, the elastic computing system determines the fastest implementation for the current situation. The elastic computing system determines the fastest implementation for an elastic function by computing the estimated execution time of each candidate implementation and selecting the fastest. Only the implementations that support the corresponding elastic function interface and that can execute on a subset of the available resources are considered (e.g., the elastic computing system cannot select an FPGA implementation if an FPGA is not available). For each of those implementations, the corresponding performance profile is used to estimate the execution time given the actual input parameters of the interface's invocation. If a performance profile does not contain a significant point that corresponds to the current invocation parameters, the elastic computing system uses linear interpolation to estimate the execution time.

After selecting the fastest implementation, the elastic computing system executes the implementation within its own execution context, which defines the group of resources allocated for that implementation. The resources are categorized into two types: primary and secondary. Primary resources are controlled by the elastic computing system and are instructed of which implementation to execute. With the current version of the elastic computing system, only CPU's can be primary resources, although the framework could be extended to consider other resources. Secondary resources (e.g., FPGA's) are associated with the execution context but are not directly controlled. Instead, the implementation itself, running on the primary resources, has the option to take control of any secondary resources within its same execution context (e.g., an implementation running on a CPU instructing an FPGA to perform some computation).

Before executing a selected implementation, the elastic computing system first sets up the execution context and then starts the implementation's execution on all of the primary resources. Each primary resource executes the same implementation. All of the invocation parameters are passed into the implementation, as well as an extra parameter that specifies the execution context. It is through the execution context parameter that the different instances of the implementation can communicate with each other. In this way, the implementation running on the primary resources are similar to a single-instruction-multiple-data (SIMD) function. Additionally, the execution context is similar to an MPI Communicator.

When dependent implementations internally invoke an elastic function call, the implementation has the option to partition the execution context into multiple sub-contexts. Each sub-context is independent of each other and can execute different elastic functions and/or pass different parameters. It is through this mechan-

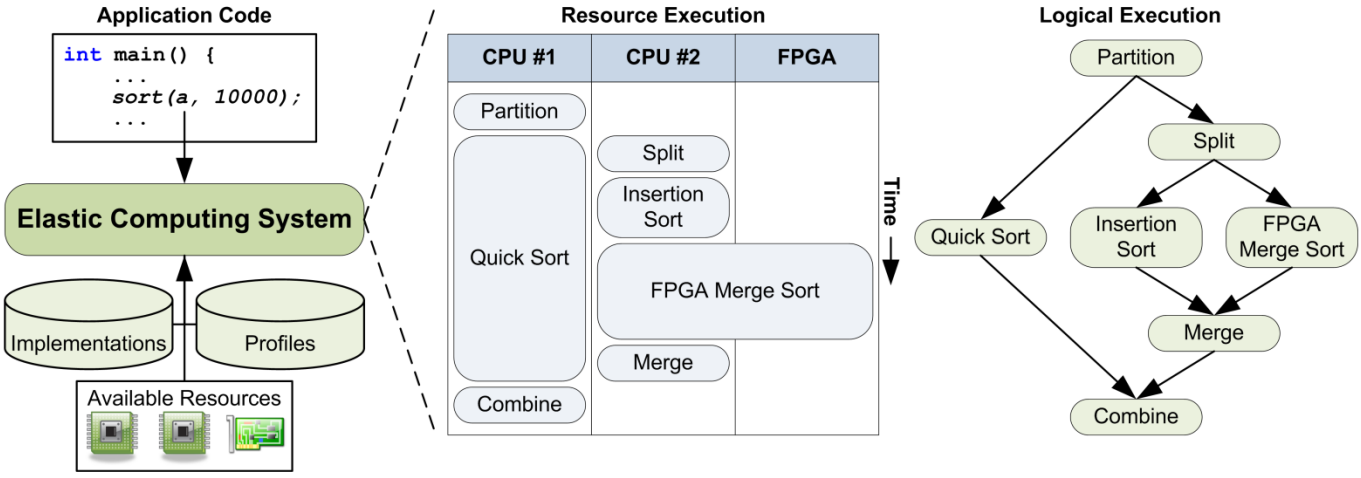

Figure 5. When an application executes an elastic function, the *elastic computing system* uses the performance profiles from implementation planning to select and execute the fastest implementation for the available resources and input parameters.

ism that the elastic computing system allows task-level parallelism.

The elastic function is complete when the outermost implementation finishes. At this point, the elastic computing system returns control to the originating application, which resumes execution.

Figure 5 illustrates an example of the elastic computing system, showing the independent implementations selected for a sorting elastic function (details regarding execution contexts and dependent implementations are omitted for brevity). When the application executes the **`sort()`** interface function, the interface invokes the elastic computing system, which searches for the fastest implementation for an input size of 10,000 and available resources consisting of two CPU's and one FPGA. The elastic computing system uses the performance profiles of the candidate implementations and determines that the fastest implementation is a dependent implementation. The dependent implementation first partitions the input using a CPU and then invokes two additional elastic sorting functions, executing in parallel, to sort the partitioned data. For one of those functions, the elastic computing system determines the fastest implementation is a Quick Sort implementation. For the other function, the elastic computing system selects another dependent implementation that first splits the data and then uses Insertion Sort running on a CPU and Merge Sort running on an FPGA to sort the split data. The elastic computing system then selects a Merge implementation to merge the results from the Insertion Sort and FPGA Merge Sort. Finally, the outer-most dependent implementation executes a Combine implementation to combine the results into the final sorted output. The logical execution, ignoring timing, is shown on the right side of the figure. Note that all of the information used to determine the fastest implementation is saved in the performance profiles from the implementation planning step.

6. Usage Scenarios

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. [21]. Although elastic computing improves upon previous approaches by using multiple implementations, assuming that appropriate elastic function libraries are available, application designers do not need to create these implementations. In the ideal case, application designers would use elastic functions in the same way as existing, widely-used function libraries.

Of course, there may be situations where new implementations need to be created either to target new hardware or provide new functionality. Elastic computing aids the development of new implementations by providing an environment for implementation execution and allowing an implementation to call existing elastic functions. As discussed in Section 5, the elastic computing system provides communication and resource management features to the implementation. Additionally, as most implementations can be broken up into simpler steps (e.g., a convex hull implementation that internally relies on sorting), the implementation could be coded to internally call pre-existing elastic functions, thereby allowing even the individual steps to benefit from elastic computing. Lastly, elastic computing provides a framework for code reuse by allowing any developed implementation to be simply included as another implementation option for an elastic function. As a result, the more pervasive elastic computing becomes, the less often new implementations or elastic functions will need to be created.

We envision that elastic function libraries could potentially be created for different applications domains, where implementations (and corresponding adapters) of each elastic function are provided by several potential sources. Device vendors for specialized devices are one likely source for implementations, because by enabling transparent usage of their corresponding devices via elastic functions, those devices could potentially become useable by new markets (e.g., mainstream designers). For example, Xilinx could attract software designers by enabling transparent usage of their FPGA's via elastic function implementations for functionality common to specific domains. Third-party library designers such as Rapidmind [20], who already target specialized devices, could also provide implementations of elastic functions for numerous devices. Finally, open-source projects 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 the elastic computing system to automatically identify fast implementations, mainstream application designers could transparently exploit an implementation specialized for any given situation without any knowledge of the actual implementation or situation.

6.1 Summary of Advantages

Transparency Elastic computing achieves transparency of implementation and architecture, enabling non-expert designers to take advantage of powerful multi-core heterogeneous systems. Elastic computing also achieves a more transparent integration into existing tool flows compared to new language approaches that have largely been resisted due to the inconvenience of modifying well-established tool flows. In most cases, taking advantage of elastic computing is a simple matter of replacing pre-existing function calls with their elastic function equivalents.

Portability Unlike existing applications, which without modifications typically do not improve in performance when executed on a system with more or different types of resources, elastic computing invisibly selects/creates implementations that can utilize extra resources to improve performance.

Adaptability Elastic computing can adapt the implementation of a function to runtime parameters and changes in resources. Elastic computing has the flexibility to take advantage of as many or as few computing resources as it deems would be most efficient. For embedded systems, elastic computing could be extended to automatically choose power-efficient implementations and avoid using certain computing resources when battery life is low. For space systems, elastic computing could potentially avoid using computing resources that have been flagged as being damaged or disabled. Previous work has implemented similar behavior manually, but has required significant designer effort and device expertise.

6.2 Summary of Limitations

The main limitation of elastic computing is that improvement in 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.

Potential limitations of implementation planning include the time required for implementation planning to complete and the accuracy of the resulting performance profiles. Implementation planning requires several executions of each implementation which may require a considerable amount of time, but this process is only required once for a system and can be amortized over its entire lifetime. Implementation planning creates performance profiles by empirically measuring the execution time of implementations executing on sample input data. The empirical measurements reflect system effects, such as cache size and memory latency, but cannot account for some run-time issues, such as non-representative input data and resource contention. However, these effects are normally not significant enough to affect run-time performance as the elastic computing system does not make decisions based on absolute execution time (i.e., "how long will this implementation take?") but only relative execution time (i.e., "which implementation is the fastest?"). None the less, reducing these errors is an on-going research challenge.

Another potential limitation is the runtime overhead of implementation selection by the elastic computing system. As shown in the results, the performance improvement of using an elastic function normally greatly outweighs the overhead of dynamically determining its implementation. For elastic functions that require a very short execution time, the elastic computing system will likely immediately decide on an independent implementation, requiring the overhead of only a single implementation selection. None the less, repeated calls to elastic functions with extremely quick execution times could be dominated by the implementation selection overhead. For these cases, future work focuses on compile-time analysis of the source code to replace those elastic function calls with a pre-selected implementation, eliminating all runtime overhead.

7. Experiments

7.1 Experimental Setup

To perform elastic computing experiments, we developed the described implementation planning and elastic computing system tools in addition to several elastic functions. In total, over 11,000 lines of C++ code were required.

We implemented a *Sort* elastic function, in addition to nine others described later, with the following independent implementations: static size-2/3/4 sort network, in-place/out-of-place insertion-sort, heap-sort, quick-sort, and an FPGA-based in-place/out-of-place merge-sort. The following dependent implementations were also created: in-place/out-of-place parallel/serial merge-sort (i.e., two sorts followed by merging the results) and in-place/out-of-place parallel/serial quick-sort (i.e., a partition followed by two sorts). To enable these dependent implementations, we also created independent implementations for elastic functions to perform a partition (first-step of a quick-sort implementation) and a merge (last-step of a merge-sort implementation). All microprocessor-based code was written in C++ and compiled using g++ 3.4.4 with –O3 optimizations. All FPGA-based code was written in VHDL and compiled using Xilinx ISE 9.2i.

We evaluated the elastic computing framework on three diverse systems. The first was a 3.2 GHz hyper-threaded Intel Xeon microprocessor with a Nallatech H101-PCIXM FPGA accelerator [24], which has a Xilinx Virtex IV LX100. The second was a 2.4 GHz 16-CPU AMD Opteron 880 system (8 dual-core microprocessors) with no FPGA's. The third was a 2.4 GHz quad-core Intel Xeon also with no FPGA's.

All results represent total execution time, including elastic computing overhead, communication times, etc. The only overhead not included is the configuring of the FPGA with the bitfile, as all VHDL code was compiled into a single bitfile that was preloaded at application startup. The baseline comparisons were written using hand-optimized serial code to represent a fair alternative to using elastic functions when targeting an arbitrary system. To ensure accuracy, each result is the average of several executions.

7.2 Results

Figure 6 illustrates the portability advantages of elastic computing for the *sort* elastic function with an input of 4,194,304 elements, running on the 16-CPU Opteron system. The baseline is a Quick Sort function running on a single CPU. As more resources are allocated, the elastic function automatically takes advantage of the additional computing power without any coding changes, resulting in a speedup of over 4x for 16 CPU's.

Figure 7 illustrates similar advantages of several elastic functions for embedded systems, each of which includes microprocessor and FPGA-based independent implementations, running on the Xeon/Nallatech system. *Sort* is the previously described sort elastic function with an input of 4,194,304 elements. *Prewitt* performs Prewitt edge detection on a 640x480 pixel image. *Conv* (convolution) convolves a 1 million length vector with a 256 length vector, with each element being single-precision floating-point. *MM* (matrix multiply) multiplies two 1024x1024 matrices

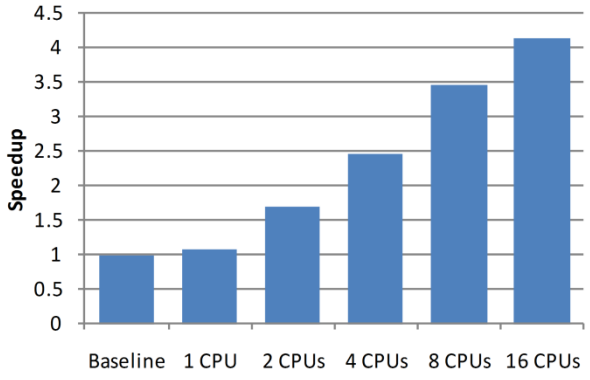

Figure 6. Portability of a *Sort* elastic function, illustrated by increasing speedup when allocated additional resources.

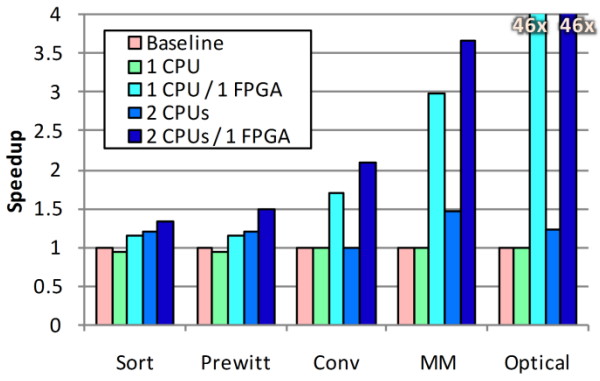

Figure 7. Speedups illustrating elastic computing portability for several elastic functions running on a hyper-threaded Intel Xeon system with a Nallatech H101-PCIXM FPGA accelerator.

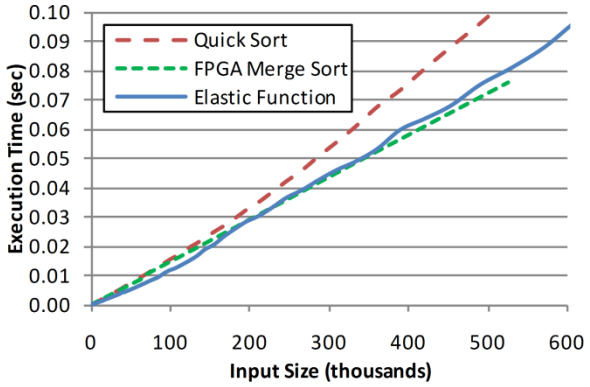

Figure 8. Elastic computing adaptability for a hyper-threaded Intel Xeon system with a Nallatech H101-PCIXM FPGA accelerator, showing that for all input sizes, elastic computing achieves the best (or near best) implementation.

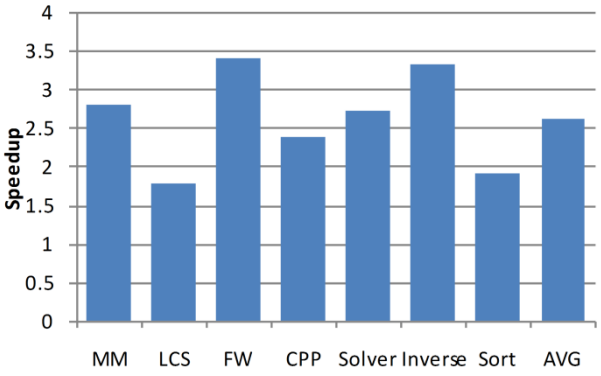

Figure 9. Elastic computing speedup on a quad-core Intel Xeon system for several elastic functions, compared to serial baselines.

using single-precision floating-point. *Optical* (optical flow) creates a two-dimensional match statistic map for a 17x17 template in a 640x480 pixel image. In all cases, elastic computing enables the application to gain significant speedup, ranging from 1.3x to 46x, by transparently taking advantage of the multiple cores and FPGA. The large 46x speedup of the *Optical* implementation is due to the FPGA exploiting a significantly larger amount of parallelism compared to the other examples.

Figure 8 illustrates the adaptability of elastic computing by comparing the performance between the independent implementations of sort and the elastic function. The results are shown across different input sizes running on the Xeon/Nallatech system. For all possible input sizes, elastic computing either achieved the fastest implementation or had a minimal overhead. For input sizes greater than 512k, elastic computing significantly out-performed the individual implementations. Note that the FPGA Merge Sort implementation only supports input sizes less than 512k elements due to limited on-board memory; however, the elastic function can still make use of it with larger input sizes by relying on dependent implementations to partition larger input sizes into multiple sub-sorts.

Figure 9 illustrates speedups of several additional elastic functions, chosen from graph analysis, computational geometry, and linear algebra, running on the quad-core Xeon system, compared to serial baselines. For each elastic function, we created a parallel independent implementation. *MM* (matrix multiply) multiplies two 1000x1000 matrices using single-precision floating-point. *LCS* (longest common subsequence) determines the longest common subsequence between two 2,500 character strings. *FW* (Floyd-Warshall) finds the shortest path between 1,000 vertices in a weighted directed graph. *CPP* (closest-point pair) determines the closest pair of points from 1,000,000 points in a two-dimensional space. *Solver* solves a system of 1,000 linear equations. *Inverse* inverts a 1000x1000 floating point matrix. *Sort* is the previously described sorting function with 4,194,304 elements. In all cases, elastic computing transparently selected/created parallel implementations for each elastic function, resulting in speedup that ranged from 1.8x for *LCS* to 3.4x for *FW*, achieving an average speedup of 2.6x.

Note that although these speedups were obtained by using extra resources, the significance of elastic computing is that *no coding modifications were required to use those resources*. To our knowledge, no previous study has shown improved performance for such diverse systems without manual code modifications.

8. Conclusions

In this paper, we introduced a framework for 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 func-

tionality in terms of elastic functions, which the elastic computing framework converts into specialized implementations through a combination of implementation planning and the elastic computing system. We evaluated elastic computing on three 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 run-time parameters, such as input size, achieving performance significantly better than the individual implementations, even with the overhead of the elastic computing system. Overall, elastic computing achieved significant speedup, ranging from 1.3x to 46x, without any changes to the application code and without any designer effort. Future work includes evaluation of different resources (e.g., GPU's), reduction of run-time overhead for elastic function calls with input parameters partially known at compile-time, and implementation planning for power consumption.

Acknowledgments

This research was supported by the National Science Foundation (CNS-0914474).

References

[1] J. Ansel, C. Chan, Y.L. Wong, M. Olszewskim, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2009, pp. 38-49.

[2] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. International Journal of High Performance Computing Applications, Vol. 21, Issue 3, August 2007, pg. 291-312.

[3] W. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick. A Performance Analysis of the Berkeley UPC Compiler. Proceedings of the International Conference on Supercomputing (ICS), 2003, pg. 63-73.

[4] Cray, Inc. Cray XT5 System. 2008. http://www.cray.com/Products/XT/Product/Technology.aspx.

[5] A. DeHon. The Density Advantage of Configurable Computing. Computer, Vol. 33, Issue 4, April 2000, pp 41-49.

[6] ElementCXI, Inc. ECA-64. http://www.elementcxi.com/productbrief.html.

[7] A. Fin, F. Fummi, and M. Signoretto. SystemC: A Homogenous Environment to Test Embedded Systems. Proceedings of the International Workshop on Hardware/Software Codesign (CODES), 2001, pp 17-22.

[8] M. Frigo and S. Johnson. FFTW: an Adaptive Software Architecture for the FFT. Acoustics, Speech and Signal Processing. Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, 1998, pp. 1381-1384.

[9] M. Girkar and C. Polychronopoulos. Extracting Task-Level Parallelism. ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 17, Issue 4, July 1995, pp. 600-634.

[10] B. Grattan, G. Stitt and F. Vahid. Codesign-Extended Applications. IEEE/ACM International Symposium on Hardware/Software Codesign (CODES), 2002, pp. 1-6.

[11] Z. Guo, W. Najjar, F. Vahid, and K. Vissers. A Quantitative Analysis of the Speedup Factors of FPGAs over Processors. Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA), pp. 162-170, 2004.

[12] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations. Proceedings of International Conference on VLSI Design (VLSI), 2003.

[13] H. Peter Hofstee. Power Efficient Processor Architecture and the Cell Processor. Proceedings of the International Symposium on High Performance Computer Architecture (HPCA), 2005, pg. 258-262.

[14] B. Holland, K. Nagarajan, C. Conger, A. Jacobs, and A. George. RAT: a Methodology for Predicting Performance in Application Design Migration to FPGAs. Proceedings of the Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA), pp 1-10, 2007.

[15] Intel Quad-Core Xeon. 2008. http://www.intel.com.

[16] L. Lewins and K. Prager. Experience and Results Porting HPEC Benchmarks to MONARCH. Proceedings of Workshop on High Performance Embedded Computing (HPEC), 2008.

[17] C. Luk, S. Hong, and H. Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO), 2009, pg. 45-55.

[18] M. Macedonia. The GPU Enters Computing's Mainstream. IEEE Computer, Vol. 36, No. 10, October 2003, pp. 106-108.

[19] I. McCallum. Intel QuickAssist Technology Accelerator Abstraction Layer (AAL) 317481-001US. 2007. http://download.intel.com/technology/platforms/quickassist/quickassist_aal_whitepaper.pdf.

[20] M. D. McCool. Data-parallel programming on Cell BE and the GPU using the Rapidmind development platform. In GSPx Multicore Applications Conference, 2006.

[21] S. Merchant, B. Holland, C. Reardon, et al. Strategic Challenges for Application Development Productivity in Reconfigurable Computing. Proceedings of the IEEE National Areospace and Electronics Conference (NAECON), 2008.

[22] K. Morris. FPGAs in Space: Programmable Logic in Orbit. FPGA and Structured ASIC Journal, August, 2004.

[23] D. Musser. Introspective Sorting and Selection Algorithms. Software: Practice and Experience, Vol. 27, Issue 8, 1999, pp. 983-993.

[24] Nallatech Inc. Nallatech PCIXM FPGA accelerator card, 2008. http://www.nallatech.com/?node_id=1.2.2&id=41.

[25] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox. Pace—A Toolset for the Performance Prediction of Parallel and Distributed Systems. International Journal of High Performance Computing Applications, Vol. 14, No. 3, 2000, pp. 228-251.

[26] L. Semeria, K. Sato, and G. De Micheli. Synthesis of Hardware Models in C with Pointers and Complex Data Structures. IEEE Transactions of Very Large Scale Integration Systems (TVLSI), Vol. 9, Issue 6, December 2001, pp. 743-756.

[27] G. Stitt, F. Vahid, and W. Najjar. A Code Refinement Methodology for Performance-Improved Synthesis from C. Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2006, pp. 716-723

[28] Tilera Tile64 Processor Family. 2008. http://www.tilera.com/products/processors.php.

[29] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. Journal of Physics, June 2005.

[30] R. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. Proceedings of ACM/IEEE Conference on Supercomputing (SC), 1998, pp. 1-27.

[31] J. Williams, A. George, J. Richardson, K. Gosrani, and S. Suresh. Fixed and Reconfigurable Multi-Core Device Characterization for HPEC. Proceedings of Workshop on High-Performance Embedded Computing (HPEC), 2008.

[32] Xilinx Inc. Virtex IV FX devices, 2008. http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm.