**REIMAGINING HETEROGENEOUS COMPUTING: A FUNCTIONAL INSTRUCTION-SET ARCHITECTURE COMPUTING MODEL**

The authors demonstrate how the functional abstraction level determines the capability and variety of a processor’s functional units and accelerators, thereby restricting its degree of heterogeneity. Combining current heterogeneous techniques with software abstraction concepts, the authors propose a new functional instruction-set architecture (F-ISA), which raises the functional abstraction level of machine instructions and offers greater heterogeneity, resulting in latency, memory footprint, and power/performance gains.

Daniel Nemirovsky  
Nikola Markovic  
Osman Unsal  
Mateo Valero  
Adrian Cristal  
Barcelona Supercomputing Center

In an innovative step forward, the hardware community has started to tackle the power wall and the memory wall by diversifying the computational cores. Whereas initial chip multiprocessors (CMPs) integrated several identical computation cores per chip, known as homogeneous processors, we now see an increasing tendency to explore the integration of diverse computational cores, called heterogeneous processors. Heterogeneous processors are divided into two classifications. The first is a processor that includes two or more computing cores that rely on the same instruction-set architecture (ISA) but are microarchitecturally different—for example, a processor that includes two MIPS ISA cores, one out-of-order four-way superscalar core, and one more simplistic in-order core. These types of heterogeneous cores are commonly called asymmetrical CMPs. Although they are just beginning to gain traction within the industry, researchers have investigated their potential.1 The second type of heterogeneous processor comprises multicore chips based on two or more different ISAs—for example, a processor than includes one MIPS computing core and one ARM-based computing core. The IBM Cell processor is an actual case of such a heterogeneous processor,2 as is ARM’s big.LITTLE.3 Consequently, processors that
offer both types of heterogeneity show great potential.

On the software front, the standardization of programming practices has promoted the development of vast amounts of libraries, tools, and frameworks. This has marked an unprecedented growth in the level of abstraction available for the average programmer. Emphasis on code optimization and portability has led to the development of programming models and runtime systems that let programmers define and extract substantial amounts of parallelism in their code, such as OpenCL (www.khronos.org/opencl), Cilk Plus (www.cilkplus.org), and OpenMP. Used effectively, these programming models allow significant gains in application performance and resource utilization. Yet, while these software models have been useful and complementary to advances on the hardware front, their considerable runtime overheads and limited adoption have hindered their appeal and applicability. Conversely, parallel applications have been shown to share similar characteristics, which increases their appeal for standardized libraries and hardware practices.

For now, a semantic gap seems to be emerging between the advances in hardware and software. The scope and potential of new technologies have started to lead to heterogeneous many-core systems (see the “Related Work in Heterogeneous Computing Systems” sidebar), but hardware architects have not similarly embraced, as of yet, the use of high abstraction levels. This is because of the physical and compatibility constraints that hardware developers face, which are far more flexible at the software level. However, given the current state of the industry, architects need to consider radical paradigm-shifting computing-model proposals. Such proposals will not necessarily offer clear roadmaps or short-term pragmatic solutions, but they could contain the hints and alternate ideas needed to rethink the long-term vision that computer architects hope to achieve.

In this article, we address this hardware-software semantic gap by considering an unconventional computing model that merges current heterogeneous state-of-the-art hardware with the concept of abstraction that has been so useful and ubiquitous in software. To do so, we use the concept of abstraction to reinterpret the notion of the ISA. Most ISAs in use today remain (for compatibility reasons) based on designs developed several decades ago to solve that era’s physical and software constraints. The current CMP model’s scalability is inherently constrained because of the current ISAs’ functional granularity, which ensures high memory footprint and energy consumption. As a remedy, we propose a functional ISA (F-ISA) that increases the functional abstraction level of the machine instructions. Consequently, this extra level of functional abstraction enables a dramatic increase in the heterogeneous diversity of a processor’s computational units, resulting in greater specialized execution particular to the needs of the software algorithms. We hope that this alternative computational model can significantly improve and fine-tune system performance relative to latency, memory footprint, and power.

**Conceptual discussion**

Computation as we know it today is grounded on the interaction and communication between two key elements: data (to be used, manipulated, and/or produced) and functions (specifying what is done with the data). We identify three data-function computational models: data to function, function to data, and data and function.

**Data to function**

This model is embodied in a system that comprises separate memory (instruction and data) and computational structures in which, via one or more machine-level instructions, data is sent from the memory (such as DRAM) to be executed in a functional unit (such as an integer adder). This method is the standard approach used by current CPUs, and both Von Neumann and Harvard architectures fall within its scope.

This model’s limiting factor is that a functional unit’s complexity is determined by the ISAs’ functional abstraction level. An add instruction corresponds to an adder functional unit, a branch instruction to a branch unit, and so on. Because nearly all conventional ISAs rely on low-level functional instructions, the physical characteristics of the data and functional units are intrinsically
Related Work in Heterogeneous Computing Systems

There have been several developments in heterogeneous many-core systems, including MorphCore, big.LITTLE, and accelerated processing units (www.amd.com/en-us/innovations/software -technologies/apu). In addition, interesting scheduling work has touched on the foundational heuristics needed to predict and improve performance when running many concurrent threads, which are relevant to the functional instruction-set architecture (F-ISA) runtime dispatcher.

Our approach also builds on several other computing techniques and models. The Java programming environment (http://docs.oracle .com/jacase/specs/jvms/se7/html) offers a rich framework of data structures and methods that are shared as libraries and ubiquitously used in academia and industry. Java and other highly abstract cross-platform languages rely on the use of virtual ISAs (such as Java Bytecode). Existing virtual ISAs also express a very low level of functionality and must be translated into the physical processor’s ISA in order to run. This procedure naturally causes overheads, but most have been greatly minimized thanks to a continual process of optimization. The virtual instruction set computing (VISC) proposal builds on an LLVM method and applies it toward heterogeneous architectures. Although there are similarities with F-ISA—in particular, both are intended as a virtual intermediate representation—the VISC approach can be seen as complementary to F-ISA. The VISC design builds on vector parallelism as well as generalized macro dataflow graphs and, although it is novel, it sets an upper bound on the level of functional abstraction it can support from the computation cores. It also does not support function unfolding such as in F-ISA.

Other LLVM-based approaches rely on threaded-code techniques that are similar to the manner in which the F-ISA generated code is structured. However, our proposal expands this technique because the compiled code will have to reference not just F-ISA subfunctions but also other virtual or physical machine code. This means that while threaded code is deterministic (it must follow a specified call to a sub-function), our method allows a runtime to determine whether to unfold the F-ISA function or to execute a machine code implementation of the function.

The neurocomputing Galatea project is a novel proposal. By using a low-level virtual machine language that includes threaded-code techniques and a certain amount of function unfolding capabilities, it can sustain code portability across different systems (in this case, neurocomputers). This technique can also be seen as complementary to F-ISA. Although similar LLVM approaches could allow for function unfolding using threaded-code techniques, their low level of functional abstraction limits the diversity and complexity of computational cores that might be implemented, as compared with the F-ISA approach. In addition, the F-ISA proposal is intended for parallel applications running on typical workstations and mobile devices without the need to modify user code.

Task dataflow programming models provide useful abstraction tools for developers to improve the parallelization and performance of their programs. OmpSs and its earlier implementations let programmers separate sections of code into tasks that are then passed to a runtime that uses dataflow methods, including data tokens and dependency graphs, to determine which tasks are ready for execution and which cores to dispatch them for execution. Whereas these programming models involve the developer inserting hints or pragmas into the application code, our approach aims at optimizing the hardware without requiring any modifications of the user code.

Nonconventional object processors have previously sought to exploit the modularity and parallelism inherent in object-oriented software techniques. The Intel iAPX 432, Rekursiv, and Smalltalk on a RISC (SOAR), are all examples of early, yet novel, object-oriented processors. More contemporary designs include the pic0Java and TinyJ (http://people.cs.uchicago.edu/dinoj/smart-card/tinyJ.html) processors based on a hardware implementation that can run Java.

The limited diversity and complexity of the functional units mean that they can understand only primitive data objects (integers, doubles, and branches). Consequently, this produces a homogenizing effect on the executable data objects and furthers the need for repetitive executions to perform higher-complexity functions (such as list sort and matrix multiplication). These factors can significantly contribute to a system’s overall memory footprint and power usage, a problem that is further exacerbated by the fact that conventional CPUs consolidate the available functional units into a handful of computational cores. Compared with a heterogeneous core configuration, in which each core contains specialized functional units suited to specific data structures, memory contention is greater when all cores are equally capable of executing the same functions on the same data.

Function to data

In this model, functional machine-level instructions are sent to a unified (that is, homogeneous) memory structure that contains both data and functional units, such that the execution happens directly within the memory structure. A key limiting factor
of this model is that in order to maintain compatibility and reliability, all memory structures and substructures must contain access to the same set of functional units. A variation of this model is used occasionally to complement standard CPU processing. Processing in memory technologies and other vector functional units directly built in memory are examples of such cases.

Data and function

This is a hybrid approach that is similar to the function-to-data model but allows for separate and distributed (that is, heterogeneous) memory structures. It is therefore possible to separate different data objects (such as matrices from lists) into separate memory structures that contain the functional units specific to those data objects (for example, the matrix multiplication unit versus the list sorting unit). Although this approach is elegantly promising, its principal drawback is that the size and composition of the physical memory structures should correspond with the data structure itself. Any slight modification or extension to an existing data object could result in significant performance penalties. This is because changes in the data object can affect not only the functional part of the execution but also the organization and quantity of data that the object uses. Moreover, if this radical approach is implemented...
directly, it will create considerable code portability and compatibility complications. Although there seem to be interesting developments in implementing this model using specific accelerators and FPGAs, research in this area is only just starting to scratch the surface of its true potential.

To expand the hardware’s heterogeneous diversity, we must rely on one of the three computational models discussed. The feasibility of the function-to-data scheme is lacking, because the need to replicate every type of functional unit for every memory structure is unrealistic and unreasonable. Variations of this model, however, offer clear opportunities for some fields of application, most notably embedded systems and graphical processing. On the other hand, we can reconsider using the standard data-to-function model. But as we noted, if we were to use this approach by replicating the number of cores on a chip, we would hit an impasse. If we could extract enough parallelism from the workloads to keep the system constantly fed and balanced, our progress would be halted by the memory wall and dark silicon, whereas if we underfed the system, we would be inefficiently using the available transistor richness. The data-and-function model could perhaps offer the most benefit in terms of specialization based on a particular workload or programming model, but its structural inflexibility and code portability and compatibility concerns must be overcome. However, we could attenuate these concerns by raising the ISA's functional abstraction level. This could allow for a substantially greater diversity and quantity of functional units, while also providing flexibility in how the code is executed.

**Functional ISA**

Current ISAs, which most conventional CPUs are designed to execute, provide low levels of functionality that limit the physical functional units’ scope and diversity. Far from considering them ineffective or wanting to replace such instrumental CPUs, we seek to foster code portability and compatibility using a flexible model and applying it gradually. Thus, we developed a higher functional-level intermediate ISA based on a conceptual combination of hardware principles and software abstraction techniques.

Similar to typical ISAs, our F-ISA consists of instructions that define a functional method and a data element. However, this F-ISA follows a top-down approach, starting from a software perspective, to determine the functionality and data context of each particular F-ISA instruction. A program is typically developed using one or more abstraction levels (that is, using libraries and classes of classes), which, when unfolded—for instance, by a compiler—expose lower and lower abstraction levels until only pseudo machine code (such as LLVM) is left. The theoretical objective of the F-ISA is to capture and follow each of these abstraction levels for as many function or method instances as possible. Practically speaking, F-ISA’s appeal is its applicability in commonly used libraries and frameworks that have functions, objects, and methods found across different programs. An example of such a ubiquitously used library is the Java standard library, which includes object structures and methods corresponding to vectors, hash tables, and lists. Mobile applications are valuable, and more recent examples in which common data structures, functions, and methods (belonging to iOS or Android application programming interfaces) are frequently used across different applications that still rely on conventional general-purpose CPUs for execution.

Compared to the composition of a typical ISA, the structure of F-ISA is flexible and open-ended so that its final composition can be adapted on a per-application basis. The end representation of a program as F-ISA code is analogous to a call graph, in which each node represents one particular functional instruction. Every instruction consists of three elements: an instruction identifier, a function identifier, and a data context address. The instruction identifier is a unique address associated with the memory location where the F-ISA instruction can be found, similar to a typical instruction’s PC address, and is needed to preserve code sequentiality. The function identifier is similar to a traditional opcode and denotes the specific executable function that the instruction will request (for example, MATMUL would be an identifier for matrix multiplication). Because each unique function or method in a program is assigned a particular function identifier, many more function
identifiers will be needed than there are typical
opcodes. The data context address provides the
address of the context where all the relevant
data needed to properly execute the function
can be found. To create the data context, the
data objects can be assigned particular
addresses at compile time (although they could
also be allocated dynamically), and then these
addresses are organized into a data context that
is assigned to a corresponding F-ISA instruc-
tion. We can consider the actual size of an F-
ISA instruction to be the sum of the function
identifier and the data context address. For
example, an 80-bit F-ISA instruction would
allow for a 16-bit function identifier and the
direct 64-bit address for the data context.

Example case

To illustrate how a program could be
represented as F-ISA instructions, consider
Figure 1.

Once compiled, the F-ISA representation
of this program, which has the instruction-level
syntax (function identifier, data
context address), can assume the form
shown in Figure 2.

Note that &ABC corresponds to the
address of the data context that contains the
addresses of matrices A, B, and C. Similarly,
the address &A contains the data context
address of matrix A, and so on for the other
instructions.

Each instruction defines a function that
can either directly correspond to a physical
functional unit present in the hardware (for
example, an add instruction to an adder) or
be further unfolded into the instructions repre-
senting its subfunctions. If the first method
is available, the instruction is dispatched to
the functional unit for execution, which
assumes that it will be able to most efficiently
perform the function based on the informa-
tion provided in the instruction. The second
method allows for an F-ISA instruction to be
unfolded into another set of F-ISA instruc-
tions. This is shown in the case of the

Figure 1. Pseudocode of a program comprising cross and dot product functions, and data
initialization functions for three matrices.

Figure 2. A representation of the matrix
multiply program when compiled into F-ISA
code. Each F-ISA instruction has the syntax
{ function identifier, data
context address }.
instruction \( \{ \text{func}_{\text{fill}}_{\text{data}}, \&ABC \} \), which can be broken down further into three separate F-ISA instructions: \( \{ \text{func}_{A}, \&A \} \), \( \{ \text{func}_{B}, \&B \} \), and \( \{ \text{func}_{C}, \&C \} \). This unfolding process can be continued for each instruction until the first method (that is, when the level of functionality of the instruction matches that of a hardware functional unit) can be applied. If the function ‘main()’ marks the highest functionality level that an F-ISA instruction can specify, the lowest is equivalent to the functional level expressed in conventional ISAs. For practical reasons, an F-ISA instruction can also be expressed as a routine comprising a set of machine-level instructions (that is, typical ISA instructions). This feature allows for the preservation of conventional ISA processors because any F-ISA instruction can be sent to a typical computational core as a machine-level routine. Additionally, an F-ISA instruction’s ability to be expressed either as a collection of machine-level instructions or unfolded into lower-level functional instructions allows for gradual implementation and compatibility.

Given that the same program should be able to run on different physical systems, which could each consist of different computational cores and functional units, a specialized hardware or software runtime will be needed to determine whether to unfold or dispatch the F-ISA instructions to the appropriate and available functional units. Although a detailed description of such a runtime falls outside the scope of this work, we have been researching and designing possible hardware implementations. Figure 3 shows the steps a theoretical F-ISA runtime unit would take when determining how to process each F-ISA instruction.

To most effectively take advantage of the program just presented, the hardware should have a matrix accelerator that includes functional units specifically designed to efficiently execute matrix-based operations, such as dot and cross products. Furthermore, because
most, if not all, matrix object data should be stored close to the accelerator, there would be lower memory bandwidth contention due to reduced data movement and cache conflict and capacity misses within the processor. Therefore, combining the use of F-ISA instructions with specialized functional units or accelerators could enable significant latency, power, and memory footprint performance benefits.

At this point, we do not expect to present a comprehensive and faultless conceptual model. Because this is a new and unconventional proposal, various challenges must be met to strengthen the model’s practical realization. Some key topics that we need to elaborate further include runtime and compiler support, memory management (including data allocation and mapping), data dependencies, program sequentiality, and I/O and OS exception and interrupt handling. Although certain existing techniques, including dynamic compilation, dynamic memory allocation, and dataflow runtime models, could help alleviate some of these concerns, we will undoubtedly have to develop new schemes to strengthen the proposed model.

Our near-term goals include expanding the F-ISA computational model to consider memory allocation and mapping concerns, developing a detailed analytical model of an F-ISA system, studying runtime and compiler support, and compiling a set of microtestbenches into a F-ISA representation and simulating their execution.

The vast increases in transistor densities on chips have offered a challenge to computer hardware architects. Keeping in mind the power limitations of uniprocessor designs, architects have responded by integrating several optimized computational cores within one processor. The push into embedded computing has added further latency and power constraints, resulting in an increasing interest in heterogeneous many-core processors. However, the current trend of exploiting heterogeneity is inherently limited in scope by the low functionality level provided by the ISAs of existing processor architectures. This results in a narrow set of possible functional units that can be used and also serves to homogenize the data, which, for good reasons, had been meticulously distinguished at the software level. This work offers a new and unconventional computing model that raises the level of functional abstraction of the hardware instructions to enable greater flexibility and diversity for implementations of hardware functional units and accelerators. This method can enable significant advances in relation to object data mapping and execution, resulting in latency, memory footprint, and power/performance gains. The hope of our long-term vision is to support a powerful heterogeneous many-core processor that symbolically embodies the diversity and functionality found in a Swiss army knife.

References

Daniel Nemirovsky is a PhD candidate at the Universitat Politècnica de Catalunya and a researcher in the Computer Architecture for Parallel Paradigms Group at the Barcelona Supercomputing Center. His research interests include high-level language architectures,
cache hierarchies, and scheduling techniques on heterogeneous chip multiprocessors. Nemirovsky has an MSc in computer architecture, networks, and systems from the Universitat Politècnica de Catalunya. Contact him at daniel.nemirovsky@bsc.es.

Nikola Markovic is a PhD candidate at the Universitat Politècnica de Catalunya and a researcher in the Computer Architecture for Parallel Paradigms Group at the Barcelona Supercomputing Center. His research focuses on hardware and software design for high-performance multithreaded computing on symmetric and asymmetric chip multiprocessors. Markovic has an MSc in computer architecture, networks, and systems from the Universitat Politècnica de Catalunya. Contact him at nikola.markovic@bsc.es.

Osman Unsal is a group manager in the Computer Architecture for Parallel Paradigms Group at the Barcelona Supercomputing Center and a researcher at the BSC-Microsoft Research Centre. His research interests include computer architecture, fault tolerance, and transactional memory. Unsal has a PhD in electrical and computer engineering from the University of Massachusetts, Amherst. Contact him at osman.unsal@bsc.es.

Mateo Valero is a full professor in the Computer Architecture Department at the Universitat Politècnica de Catalunya and the director of the Barcelona Supercomputing Center. His research focuses on high-performance computer architectures. Valero has a PhD in telecommunications from the Universitat Politècnica de Catalunya. He is a fellow of IEEE and the ACM and is an Intel Distinguished Research Fellow. Contact him at mateo.valero@bsc.es.

Adrian Cristal is a researcher and comanager of the Computer Architecture for Parallel Paradigms Group at the Barcelona Supercomputing Center and the Artificial Intelligence Research Institute, Spanish National Research Council. His research focuses on microarchitecture, multicore architectures, and programming models for multicore architectures. Cristal has a PhD in computer science from the Universitat Politècnica de Catalunya. Contact him at adrian.cristal@bsc.es.