Georgia Institute of Technology library
Call Number QA 76.758 .I579X 2004

Introduction to Morphware

Software Architecture for Polymorphous Computing Architectures

Georgia Institute of Technology
and
Space and Naval Warfare Systems Center San Diego

Version 1.0

February 23, 2004

©2004 Georgia Tech Research Corporation, all rights reserved.

This material is based in part upon work supported by the U.S. Defense Advanced Research Projects Agency (DARPA) and other agencies of the U.S. Department of Defense (DoD). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or the DoD . THE INFORMATION IN THIS DOCUMENT SHOULD NOT BE CONSTRUED AS A COMMITMENT OF DEVELOPMENT BY ANY OF THE ABOVE PARTIES.

The US Government has a license under these copyrights, and this material may be reproduced by or for the U.S. Government.

ACKNOWLEDGEMENTS

Authors

The primary authors of this document were:

Daniel P. Campbell	Georgia Tech Research Institute
Dennis M. Cottel	Space and Naval Warfare Systems Center San Diego
Randall R. Judd	Space and Naval Warfare Systems Center San Diego
Mark A. Richards	Georgia Institute of Technology

The authors wish to thank the members of the Morphware Forum who reviewed and contributed to this document. The authors also thank the Defense Advanced Research Projects Agency (DARPA) and the US Navy's Space and Naval Warfare Systems Center San Diego for their support of this work.

The Morphware Forum

The Morphware Forum is a joint activity of the participants in DARPA's Polymorphous Computing Architectures (PCA) program, as well as other interested developers of embedded computing hardware, software, and application technology. The purpose of the Morphware Forum is to define an open, portable software environment for the development of high performance applications on PCA platforms. Morphware Forum products and information are available at www.morphware.org.

The following organizations are voting members of the Morphware Forum at this writing:

- Defense Advanced Research Projects Agency
- Georgia Institute of Technology
- Lockheed Martin Advanced Technology Laboratory
- Mercury Computing
- Massachusetts Institute of Technology
- MIT Lincoln Laboratory
- MPI Software Technology, Inc.
- Protean Devices, Inc.
- Reservoir Labs, Inc.
- Stanford University
- University of Texas, Austin
- University of Southern California Information Sciences Institute

Additional contributing organizations include:

- Air Force Research Laboratory
- Applied Photonics
- BAE
- Brigham Young University
- California Institute of Technology
- George Mason University
- IBM Austin Research Laboratory
- IBM T. J. Watson Research Center
- Lockheed Martin Aerospace
- Lockheed Martin NE&SS
- Los Alamos National Laboratory
- Mississippi State University
- North Carolina State University
- Northrop Grumman
- Raytheon
- Sandia National Laboratory
- South West Research Institute
- Space and Naval Warfare Systems Center San Diego
- University of California, Berkeley
- University of California, Irvine
- University of Illinois at Urbana-Champaign
- University of Maryland
- University of Pennsylvania
- Vanderbilt University
- Vermont University
- VLSI Photonics

DOCUMENT CHANGE HISTORY

This is the initial public release.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS i
- Authors i
- The Morphware Forum i

DOCUMENT CHANGE HISTORY iii
TABLE OF CONTENTS iv
TABLE OF ACRONYMS v
INTRODUCTION 1
- Who Should Read this Document? 1
- Content of this Document 1

THE PCA PROGRAM 2
PCA SYSTEMS AND APPLICATIONS 3
- Introduction to PCA Systems 3
- Morphing 5
- Example of a PCA Application 5

PCA LANGUAGES AND COMPILATION 9
- Streaming Algorithms and Languages 9
- Two Stage Compilation 10

ELEMENTS OF THE MSI 12
- The Stable Architecture Abstraction Layer 13
- Stream Virtual Machine Overview 13
- Thread Virtual Machine Overview 14
- Metadata 15
- Machine Model Metadata Context 15

CREATING AN APPLICATION 17
FUTURE DEVELOPMENTS 19
REFERENCES 20
APPENDIX: STREAM LANGUAGE EXAMPLES 22
- Brook Example 22
- StreamIt Example 24

TABLE OF ACRONYMS

API	Application Programming Interface
ALU	Arithmetic-Logic Unit
ASIC	Application-Specific Integrated Circuit
CAT	Classification-Aided Tracking
DARPA	Defense Advanced Research Projects Agency
DSP	Digital Signal Processor
FAT	Feature-Aided Tracking
FIFO	First In, First Out
FPU	Floating Point Unit
GMTI	Ground Moving Target Indication
HAL	Hardware Abstraction Layer
HLC	High-Level Compiler
I/O	Input/Output
IRT	Integrated Radar-Tracker
ISR	Intelligence, Surveillance, and Reconnaissance
LLC	Low-Level Compiler
MIT	Massachusetts Institute of Technology
MIT/LL	Massachusetts Institute of Technology Lincoln Laboratory
MSI	Morphware Stable Interface
OS	Operating System
PCA	Polymorphous Computing Architecture
RAM	Random Access Memory
Raw	Raw Architecture Workstation
SAAL	Stable Architecture Abstraction Layer
SAPI	Stable Application Programming Interface
SAT	Signature-Aided Tracking
SVM	Stream Virtual Machine
SWEPT	Size, Weight, Energy, Performance, and Time
TFLOPS	TeraFLOPS (Tera-Floating Point Operations per Second)
TRIPS	Tera-op Reliable and Intelligently Adaptive Processing System
TVM	Thread Virtual Machine
UVM	User Virtual Machine
V&V	Validation and Verification

INTRODUCTION

Who Should Read this Document?

This document is a product of the Polymorphous Computing Architectures (PCA) program of the Defense Advanced Research Projects Agency (DARPA). It is intended for application programmers, system engineers, and managers as a high-level introduction to PCA [1] systems in general, and software development for PCA systems in particular[1]. PCA platforms offer a significant increase in capability and flexibility over traditional computing platforms. However, the systems are also significantly more complex, thus presenting new problems requiring new solutions for application development.

The PCA program established the Morphware Forum [2] to develop and establish a portable application development methodology for PCA systems. The Morphware Forum is a joint activity of the participants in the DARPA PCA program, as well as other interested developers of embedded computing hardware, software, and application technology. A methodology that includes new source languages, a new application development process, and a framework for expressing system and application metadata has been developed to allow application developers, hardware developers, and build-tool developers to deploy high performance, flexible PCA-based computing systems. Those elements of this methodology defined by the Morphware Forum are termed the Morphware Stable Interface (MSI) or simply *morphware*. The defined software, metadata, and programming standards of the MSI allow the developer to abstract diverse PCA hardware targets from the application software requirements.

Content of this Document

This document gives an overview of the elements and structure of the MSI. After reading this document an application programmer will understand the basic concepts used in developing PCA applications. Also, as an example of the type of defense applications that will benefit from a PCA system, a benchmark developed under the PCA program, the Integrated Radar Tracker, is described.

Detailed descriptions of the concepts introduced in this document are found in other MSI documents. At the time of this writing, the MSI is still under development, so the framework described here is subject to change.

[1] The acronym "PCA" is used in this document to refer variously to PCA microprocessor devices, computing systems based on these devices, and the DARPA research program. The specific meaning of each usage should be clear from the context.

THE PCA PROGRAM

DARPA's Polymorphous Computing Architectures program [1] is developing a revolutionary approach to implementing embedded computing systems that support reactive, multi-mission, multi-sensor, and in-flight retargetable missions, and that reduce the time needed for payload adaptation, optimization, and validation from years to days to minutes. The PCA program breaks the current development approach of "hardware first and software last" point solutions by moving beyond conventional computer hardware and software to flexible, "polymorphous" computing systems. A polymorphous computing system (chips, processing architecture, memory, networks, and software) will "morph" (take on or pass through varying forms or implementations) to best fit changing mission requirements, sensor configurations, and operational constraints during a mission, for changing operational scenarios, or over the lifetime of a deployed platform.

In current practice, a processor is typically selected to perform a particular class of processing, such as real-time data signal processing or, at the other extreme of the processing spectrum, cognitive reasoning. A corresponding spectrum of domain-specific processors, such as specialized data-intensive signal processors, general digital signal processors (DSP), general-purpose microprocessors, and the server-class devices, is then required to perform the complete range of mission processing. Through their ability to reconfigure resources and architectural elements to implement a broad range of architectural implementations, PCA systems can span this processing continuum with a single class of device. This is accomplished by dynamically reorganizing the PCA device's processing elements or micro-architectural components to provide an optimized architectural implementation for each specific set of mission or system requirements.

To achieve this capability, the PCA program is implementing a family of novel malleable micro-architecture processing elements to include compute cores, caches, memory structures, data paths, network interfaces, network fabrics with incremental instructions, OS, and network protocols. To support the use of these polymorphous computing systems, the program is creating a model-based software framework for reactive monitoring, optimization, modeling, resource negotiation and allocation, regeneration, and verification. A set of measurement metrics are being developed to support processing system design and optimization; these include size, weight, energy, performance, and time (SWEPT). Finally, the PCA program is establishing benchmark and standards groups that are creating community standards to enable broad application and commercial support of PCA program developments. The Morphware Stable Interface described in this document is a product of this latter activity.

PCA SYSTEMS AND APPLICATIONS

Introduction to PCA Systems

PCA devices are programmable computing devices that possess a significantly greater degree of reconfigurability than traditional general purpose computing devices. PCA devices are designed to deliver performance approaching that of Application Specific Integrated Circuits (ASICs) on a wide variety of tasks, and to rapidly adjust to meet changing, task-specific needs. This ability will increase the total efficiency of a computing platform by allowing high utilization of a large portion of the computing resources during all phases of a mission or application. For instance, in signal processing applications the resources may be configured to efficiently support data parallel operations, and then reconfigured (*morphed*) to support analysis and knowledge processing when data is available.

DARPA is supporting PCA architecture and chip development for four systems: the Raw architecture at the Massachusetts Institute of Technology [3], the Stanford University Smart Memories project [4], the MONARCH project at the University of Southern California Information Sciences Institute [5], and the TRIPS system from the University of Texas at Austin [6].

PCA devices have a pool of configurable computing resources on a single integrated circuit, connected by a configurable communication network. Computing resources consist of such elements as arithmetic logic units (ALUs), floating point units (FPUs), and various types of memory. Each resource typically can operate in one of several modes, and has some configurability within a mode. Some PCA devices support aggregation of several of the computing resources to behave as a single more complex and powerful type of resource. Computing resources are typically placed into a repeating tiled arrangement, with each tile consisting of one or more instruction processors and associated local memory. Communication networks typically consist of a fixed data path from each computing tile to external I/O and memory, as well as a configurable local path from each tile to one or more of its neighboring tiles. The notional, generic depiction of a PCA device shown in Figure 1 suggests some of these key features. Specific PCA devices, however, vary in whether they have one or multiple procesors per tile, the number and type of interconnect networks, and so forth. Although the four PCA systems supported by DARPA all feature a tiled architecture, they vary widely in their details.

Tiled computing resources with dedicated local memory increase the efficiency of processors by reducing the average distance between a processing element and the memory used by that element. With clock rates on general purpose processors constantly increasing, this distance becomes important when it exceeds the distance that memory information can travel during a clock cycle. Each of the processing cores in a tiled configuration is typically smaller and less capable than the processing core on a traditional CPU, but is able to achieve higher utilization. The presence of several such tiles on a single IC allows very high bandwidth and low latency communication between processing cores, which in turn allows applications to be parallelized more effectively

than on platforms with less efficient inter-process communication networks (*e.g.*, symmetric multiprocessors or cluster computers).

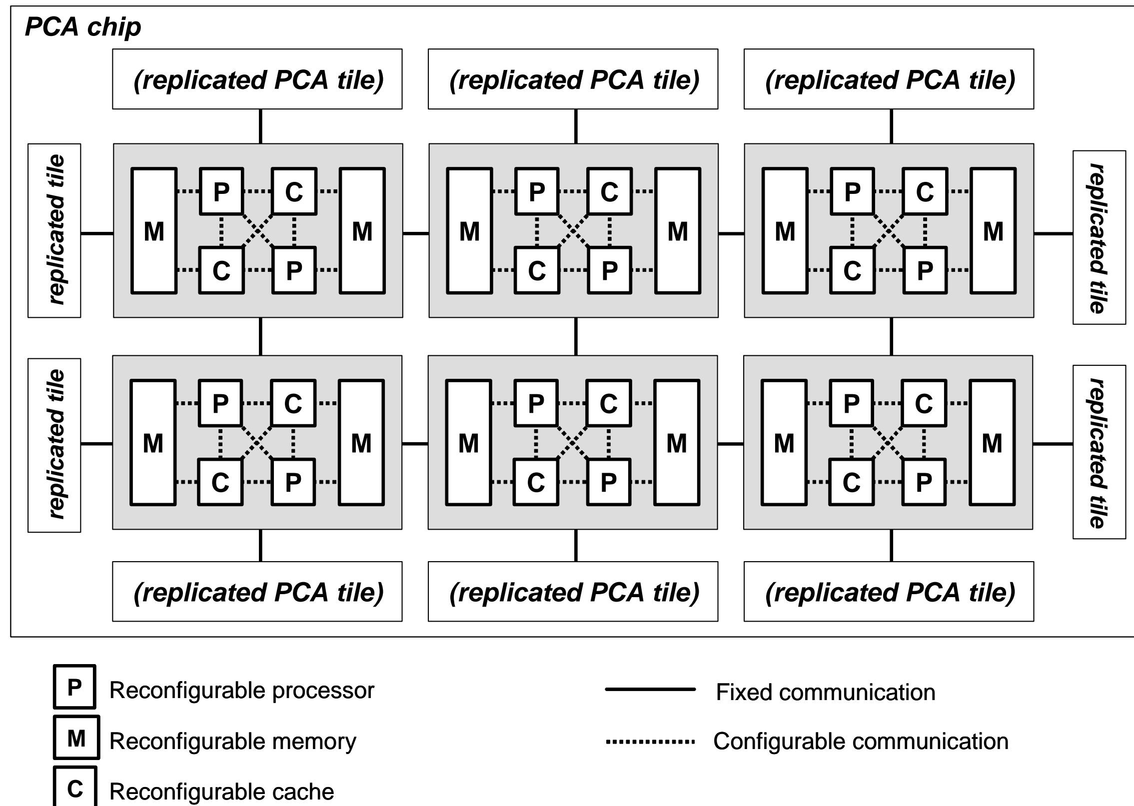

Figure 1. Generic PCA device architecture.

Two other aspects of PCA architectures contribute to their high performance. One is the high ratio of device area dedicated to computation resources such as ALUs and memory to the area devoted to control overhead. The second is the unusually high degree of control over the configuration and allocation of those resources available to the programmer and compiler.

These features allow PCA architectures to achieve a significantly higher utilization than is typical for modern traditional CPUs, for a wide variety of applications. To achieve these levels of performance, however, the application development tools must be very effective at utilizing the disparate and configurable resources present in a PCA system. In particular, the tools must be capable of identifying the parallelism in an application and partitioning that application to take advantage of the specific resources of a particular PCA chip. New languages and a new framework have been developed that allow application developers to expose data dependencies and opportunities for parallelism to the build chain, and allow the configuration space of PCA platforms to be expressed in a structured and analyzable way.

The MSI has two main goals: to reduce the effort required by tool developers, and to allow productive development of high performance portable applications. The level of effort required for tool development is reduced by standardizing and abstracting multiple

portability layers within the MSI. Creating a portable virtual machine abstraction of PCA hardware, as well as portable application level APIs, reduces development effort by presenting new tools and architectures with a common abstraction target.

Morphing

There are many possible scenarios and situations in which a change, or *morph*, in the configuration of a PCA system may be desired. The Morphware Forum has identified and categorized the types of these situations to aid in identifying the hardware and software services required by applications, operating systems, and run-time resource managers. This categorization encodes three orthogonal aspects of the attributes of a morph [7]. These are:

- whether the morph is initiated directly by an API call within the application code, or is initiated by the run-time system or compiler invisibly to the application programmer;
- whether the physical resources allocated to the application must change or stay the same; and
- whether the components of the application (or the entire application) continue to execute or are reloaded or replaced.

The set of morph types resulting from these attributes is summarized in Figure 2. At this time morph type 4a, where the platform configuration is determined by the build tools at compile time and set by the run-time environment at load time, is the only morphing type supported by the MSI. Support for additional morph types will be developed in the future.

Example of a PCA Application

Many high performance applications process data from a sensor network to solve a physical problem. Examples range from radar and sonar surveillance systems to audio and video multimedia devices. These applications can often be intuitively described by a directed graph of well-defined, computation-intensive tasks (“kernels”). Each kernel in the graph receives data from the system input or from other kernels, processes it, and passes the modified data to still other kernels or to the ultimate system output. The concept is shown in Figure 3. A common constraint of such applications is that data must pass through the graph fast enough to keep up with the real-time sensor input stream. The need for high performance, flexible implementations of applications of this type is one of the major motivators for DARPA’s development of PCA technology.

	Run-time System		Application Programmer		Compiling System	
	Components continue	**Components change**	**Components continue**	**Components change**	**Components continue**	**Components change**
Resource allocation doesn't change	*Type 0a*	*Type 1a*	*Type 2a*	*Type 3a*	*Type 4a*	*Type 5a*
	Run-time environment changes transparently to the running application.	Run-time system changes components to reconfigured but equivalent set of resources.	Application makes API call to make suggestions.	Application makes API call to change processing mode but does so within existing resource set.	Compiler instructions reconfigure allocated resources.	Compiler switches to a different library able to use the same resources.
Resource allocation changes	*Type 0b*	*Type 1b*	*Type 2b*	*Type 3b*	*Type 4b*	*Type 5b*
	Run-time system changes resource allocation of a running application transparently to the application.	Run-time system configures resources and loads components at application startup.	Application makes API call to give up or gain some resources.	Application makes API call to add or replace one or more components using different resources.	Compiler requests different resources to meet change in performance specified by metadata.	Compiler switches to a different library that uses different resources.

Figure 2. Taxonomy of Morph Types.

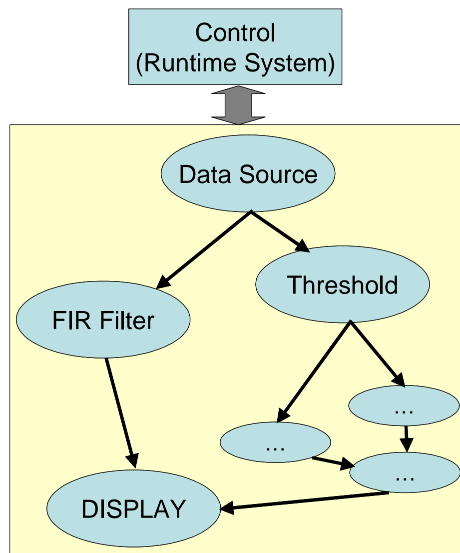

Figure 3. Illustration of the streaming computation concept. (Figure courtesy of MIT.)

To get a sense of the type and scale of a particular example of a streaming sensor application, consider the Integrated Radar-Tracker (IRT) benchmark application [8]. The IRT is an end-to-end specification of a modern intelligence, surveillance, and reconnaissance (ISR) radar system. Motivated by a space-based radar application, it embodies all of the major attributes required in a defense-oriented PCA application test: both streaming and data-dependent threaded computation with multiple sub-types of each (*e.g.*, fast transforms *vs.* vector-matrix arithmetic in the streaming elements); heavy computational loads; and multiple application-level parallelization and morphing opportunities. Developed by MIT Lincoln Laboratory (MIT/LL), the benchmark consists of a MATLAB simulation that serves as an executable specification, sample data sets, spreadsheets for estimating the computational loading of the application, and instructions for installation and operation. The benchmark is being developed in stages; as of this writing, the initial version of the IRT, which does not yet include all of the functionality described below, is available to the PCA community at the Morphware Forum web site.

Figure 4 shows a very high level view that divides the IRT application into two major blocks, ground moving target indication (GMTI) and feature-aided tracking (FAT), based on computational load and processing type. The FAT block has two options, signature- or classification-aided tracking (SAT or CAT). The total system load is a strong function of the processing parameters and the number of targets to be tracked; one set of default parameters provided by MIT/LL gives an estimated load in excess of six TeraFLOPS (TFLOPS).

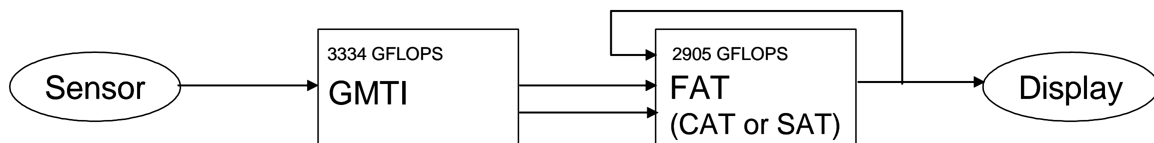

Figure 4. High-level decomposition of the IRT benchmark.

The IRT embodies both major computing types important in PCA systems, *streaming* and *threaded*. These classes of computing are described further in the next section. The large majority of the processing in the GMTI block consists of a variety of signal processing operations such as polyphase FIR filtering (for the subband analysis and synthesis), matrix-vector operations (adaptive beamforming and space-time adaptive processing), fast Fourier transforms (Doppler filtering), and correlation or convolution (pulse compression). All of these are examples of streaming operations. They apply non-data dependent, fixed (except for parameter choices) kernels to incoming data samples on a continuous or block basis, producing modified data streams or blocks that are input to the next kernel in the functional flow graph. Operations are sequenced in a dataflow manner, with each kernel able to “fire” when all of its inputs are available. Because of the fixed kernels and lack of data-dependent control flow and data production, the system can be deterministically scheduled.

In contrast, the set of operations that implement the tracking processing represented by the FAT block are highly data-dependent, with the computational load depending not only on the number of targets to be tracked, but the actual physical behavior (kinematics) of those targets. As a result, even though most of the load comes from arithmetic

calculations, the number and sequencing of those calculations varies with the sensor scenario. In addition, the tracker uses stored databases (reference signatures for a mean-squared error calculation, and a dynamic database of track histories for the kinematic tracker) as well as incoming sensor data.

The IRT can be parallelized in a number of ways, including both data-parallel and pipeline-parallel approaches, with varying levels of granularity for each. The IRT also supports application-level morphing of various granularities in both functional complexity and time scale. For instance, reconfiguring the same PCA resources from GMTI to FAT processing would require a major functional reorganization, probably with very low latency, while operator-selected parameter changes would require only a relatively minor scaling of the existing functional flow and be infrequent and tolerant of greater latency. These parallelization and morphing options provide ample opportunities to exercise PCA and MSI capabilities.

PCA LANGUAGES AND COMPILATION

The Morphware Stable Interface is composed of several novel elements. These include the use of streaming languages, a two-stage compilation with a virtual machine middle layer, and metadata to describe the target architecture and guide high-level compiler optimization. The following subsections provide more detail on each of these elements.

Streaming Algorithms and Languages

As discussed in the last section, high performance streaming sensor applications are of significant interest to the PCA program. Applications in this form may be efficiently mapped to physical resources in several ways. For instance, kernels may be multiplexed in space or time. Multiplexing in space statically assigns each kernel in the graph to disjoint physical resources. In this scheme, each kernel must be assigned to enough resources to ensure that its throughput equals to the input rate. In time multiplexing, kernels take turns using all of the system resources to process a block of input data, so long as the system is able to rotate through all the kernels quickly enough to keep up with the input data. Hybrid combinations are also possible.

Many opportunities for partitioning a streaming computing problem are masked from compilers and linkers by traditional general-purpose languages. Streaming languages are efficient on PCA hardware because they expose data parallelism and the kernel structure directly in the applications' representation in the programming language. Compilers can then determine which data may be kept local to the compute resources and streamed directly between logic units via internal buffers, greatly improving utilization efficiency, power, and throughput. Streaming languages and the associated methodology are an integral part of the MSI.

Because of the potential efficiency of streaming on PCA systems, several of the PCA hardware teams are developing new stream-based languages to aid in exploiting the capabilities of their hardware. Two of these, Brook and StreamIt, are currently supported by the MSI. While both StreamIt and Brook are similar in their use of streams and kernels to process data, they also have several significant differences. StreamIt represents a program as a single stream graph that operates on a conceptually infinite stream, while Brook supports multiple stream graphs that operate on finite streams and are controlled from a pointer-less subset of C. Kernels in StreamIt must have static input and output rates, while Brook kernels can be dynamic. StreamIt kernels can have internal state that is preserved between invocations, while Brook kernels must be stateless; also, StreamIt kernels can peek at input items without popping them from the stream, while Brook kernels must pop any items that they inspect. Finally, the stream graphs in StreamIt are composed of hierarchical units, each of which has a single input stream and single output stream, while Brook supports a flat graph of kernels, each of which can have multiple input streams and multiple output streams. StreamIt and Brook are fully detailed in their respective specifications [9,10]. Rudimentary examples of each are given in the Appendix of this document. More substantial examples of Brook code are included in Reservoir's R-Stream 1.0 high-level compiler release [11].

Two Stage Compilation

The implied abstract machine model on which traditional programming languages such as C are built has become an increasingly poor match to actual modern processor architectures. Greater use of multiple processors, multiple independent process flows within a processor, a complex memory hierarchy, and the common use of many isolated computing systems on a single application (clusters and distributed computing) have all created important deviations from the underlying computational engine for which these languages were designed. Many approaches have been taken in an attempt to maintain the usability of traditional languages, while exploiting the emergent capabilities of modern computers, including domain-specific middleware, explicit communication libraries, special codes inserted into source code to serve as compiler hints, and a litany of general guidelines and approaches for structuring programs to best suit particular platforms [12]. Each of these approaches has had some success in exploiting new technology, but has resulted in software that is tightly coupled to a particular deployment platform, and often requires a significant level of human effort to maximize performance for that configuration.

With a finite, but increasing set of source languages, APIs, and PCA target platforms, each new API or source language introduced requires an application development toolset for each possible target PCA platform, and each new PCA platform requires a development toolset for every source language and API supported. The resulting proliferation of toolsets is depicted in Figure 5.

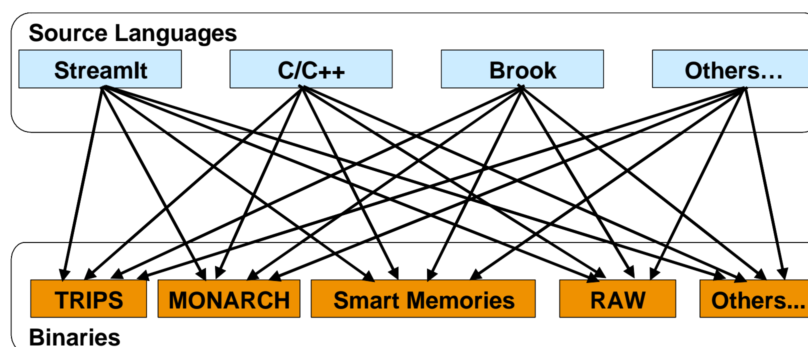

Figure 5. Multiple source languages and multiple hardware targets require development of many different software tool sets.

The approach of building multiple independent compilers and build tools is not efficient for PCA platforms, which can change and reconfigure frequently. The build-chain tools must be portable and capable of deploying an application onto a wide variety of platform configurations and in addition must be capable of selecting the optimal hardware/software configurations for a particular task and set of constraints. In order to support increased portability and to speed the deployment of new PCA devices and PCA programming languages, the Morphware Forum recognized the need to establish an explicit expression of the common abstractions of the PCA processors in the MSI. In particular, since the source languages and Application Programming Interfaces (APIs) (*e.g.*, C, C++, StreamIt, Brook) developed for PCA architectures are targeted to similar implicit abstract machines, and since PCA platforms share common abstract

characteristics, it is natural to introduce a portability layer between these APIs and the PCA hardware that encapsulates the common abstractions.

In the MSI, this portability layer is the Stable Architecture Abstraction Layer (SAAL). The SAAL is a set of portable APIs that encapsulate abstractions of the computing resources present in PCA devices, as well as the operations on those resources used by the MSI source languages and APIs. This portability layer abstracts and simplifies PCA hardware for the source languages, and provides a consistent abstract set of resource types and functional support requirements for PCA hardware developers. The SAAL portability layer also simplifies the deployment of new PCA platforms and new MSI source languages and APIs by providing a single common target for each. New languages and APIs must only provide a mapping to the SAAL portability layer, instead of build tools targeting every possible target platform. New PCA platforms need only supply a compiler for the SAAL to work with all of the existing source languages and APIs. This simplified toolset architecture is depicted in Figure 6.

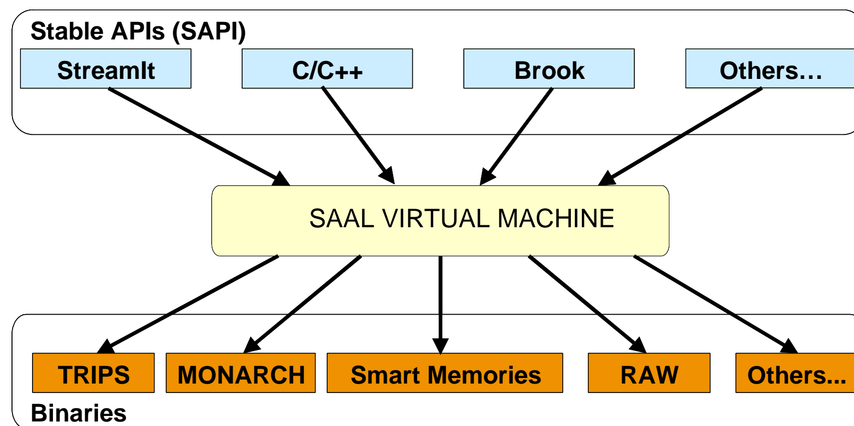

Figure 6. Introduction of a virtual machine layer reduces the number of software tools required.

The MSI framework thus uses two separate compile steps. The source languages and APIs collectively are referred to as the Stable Application Programming Interface (SAPI) layer. The top-level input at the SAPI level is processed by the high-level compiler (HLC) appropriate to the source language(s) used. The high-level compiler outputs SAPI code, which is the input to the low-level compiler (LLC) for the target PCA platform of choice. At this writing, the first alpha version of a PCA high-level compiler, Reservoir Inc.'s R-Stream, has been released [11]

ELEMENTS OF THE MSI

Figure 7 illustrates the major elements of the MSI. Applications are written in one or more of the SAPI source languages. The target platform and the set of possible configurations of the target platform are described using the PCA machine model metadata context, described below. The Morphware Forum has begun defining a metadata context for application information including such elements as performance requirements (*e.g.*, computational throughput, latency and power), input rates, and module interconnections. This metadata context is currently undefined, and no existing high-level compiler uses this context. Currently all high-level compilers optimize for execution time, and do not support specific performance requirements.

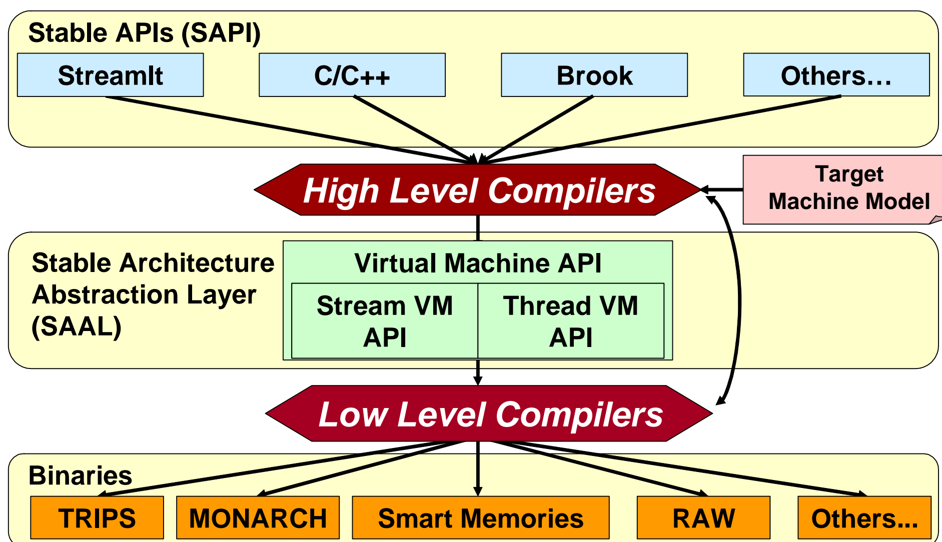

Figure 7. Elements of the MSI.

The high-level compiler is responsible for selecting the desired platform configuration, and for converting the SAPI source code to SAAL code. The high-level compiler performs coarse-grained parallelization of the input application, based on the granularity appropriate to the target platform. In order to accomplish this effectively, the high-level compiler must have a description of the target platform. This description is provided in the PCA machine model metadata context [14], where the capabilities and resources of PCA platforms are described in terms of a common, high-level model that allows the high-level compiler to perform accurate performance estimates. The output from a high-level compiler is SAAL code and a desired initial platform configuration. Depending on the application source language, the SAAL intermediate code may be Streaming Virtual Machine (SVM) code, User Virtual Machine (UVM) code, or Threaded Virtual Machine-Hardware Abstraction Layer (TVM-HAL) code, described below.

The low-level compiler is an architecture-specific build tool that accepts the SAAL code and desired platform configuration metadata output by the high-level compiler and produces native machine code suitable for execution on the target PCA platform. If a

performance constraint, resource requirement, or desired configuration is unobtainable by a low-level compiler, that low-level compiler fails and reports the error.

The Stable Architecture Abstraction Layer

The SAAL is composed of a number of parts, each addressing particular aspects and requirements of application programs and the high-level compilers. Streaming code is primarily translated into an intermediate language described by the Streaming Virtual Machine [15]. The SVM supports the mapping of kernels and streams onto virtualized processing and memory resources. Conventional code and the control code for the streaming languages are translated into an intermediate language described by the Threaded Virtual Machine. The TVM includes support for numerous non-streaming activities. A portion of the TVM, called the Hardware Abstraction Layer (TVM-HAL) [16], defines a thin abstraction layer for low-level hardware services such as memory mapping and exception handling. Many TVM capabilities, such as access to privileged instructions, are used only by operating system functions. Therefore, a User Virtual Machine (UVM) interface has been defined that contains somewhat restricted functionality that is intended to be used directly by application code and libraries [17].

Application programmers will not normally interact with any of the interfaces at the SAAL level. This code is either produced by the high-level compilers, or is contained in special-purpose libraries or operating system code written by expert library programmers. Figure 8 depicts the relationships between the various interfaces.

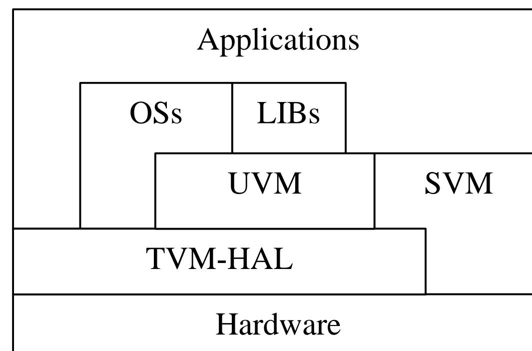

Figure 8. The relationships between applications, SAAL layer interfaces, and the underlying PCA hardware architecture.

Stream Virtual Machine Overview

The streaming virtual machine is the portion of the SAAL API that implements streaming functionality on virtual resources. The SVM allows streaming applications to be expressed in terms of operations on the set of streaming virtual resources described below. These resources are mapped directly to physical resources in a PCA system by the low-level compiler.

The virtual resources within the SVM API are:

- *Streams* are sequences of data elements of a specified type. Streams are used for sequential production and consumption of data elements.

- *Kernels* are computational engines that consume zero or more input streams, and produce zero or more output streams. Kernels are described using a subset of C, and also maintain execution state information.
- *Blocks* provide random, indexed access to a fixed set of data elements.
- *Controls* are independent computational engines that can initiate, monitor and control the state of, and terminate kernels. Controls are described with a subset of C.
- *Data elements* are primitives supported by a particular platform, arrays of fixed length, or fixed-sized `structs` containing primitives and arrays.

The SVM API is a C API that can be implemented as a library, if desired, for testing and debugging purposes. The SVM API is described in detail in the document titled “PCA Morphware Virtual Machine Specification” [15].

Thread Virtual Machine Overview

The thread virtual machine is comprised of two major elements, the TVM Hardware Abstraction Layer, and the User Virtual Machine. The TVM-HAL provides a SAAL level virtual machine interface for threaded portions of PCA applications and libraries. The TVM-HAL is a privileged interface that provides a thin layer of abstraction on the available PCA architecture and presumes total control over the available resources. The UVM is also a low-level interface, but does not presume total resource control over the platform. This allows a more direct mapping from application code, but also requires a richer interface to allow user-level response to changes in resource availability, or failure to obtain requested resources. The UVM and TVM-HAL together address the major areas of processor control, exception handling, and memory management, as well as several other low-level requirements to support general purpose threaded software.

Processor control and exception handling in the UVM and TVM-HAL include support for exception vectoring of up to 255 exceptions, with special state maintenance code for exception handlers; instructions to convert a processor from a threading mode to streaming mode; and a scheduler activation interface. The scheduler activations interface [18] provides a communication mechanism between a kernel-level scheduler and application code. This interface provides for a set of callbacks to user code when a processor has been assigned to or taken away from a process, or when a thread has become blocked or unblocked (for example, while waiting for I/O).

PCA architectures differ from conventional architectures in that they have many distinct blocks of memory distributed across the system. This requires special handling for memory addressing in the UVM and TVM-HAL interfaces. Memory management in the UVM and TVM-HAL includes support for segmented memory as well as paged memory. Paged memory control is available in the TVM-HAL but not in the UVM. The segmented memory model allows for virtual addressing and configuration of individual segments, including whether a segment is visible to a particular processor, initial virtual and physical addresses, and caching control.

The UVM and TVM-HAL support several other control constructs including active DMA, multiprocessor synchronization, mutex locks, cache memory control, performance counters, IEEE 754 floating point control, and condition variables.

Metadata

PCA systems are designed to meet a variety of goals and constraints, and to function on a wide variety of platform configurations. PCA applications require a large amount of information in addition to the procedural definitions of programs in source languages. Examples of this information are the computing resources available on a particular host platform, the set of possible configurations of these resources, desired optimization goals for a particular piece of software, and computing resources required by a particular piece of compiled software. This set of descriptive and extra-functional information is known collectively as *metadata*.

Each of the specific uses for metadata within the MSI is known as a metadata *context*. At this writing, the PCA machine model is the only defined metadata context. Others are under consideration by the Morphware Forum, for example an application information metadata context and a context describing high-level compiler output assumptions and decisions.

In order to ease the development of build-chain tools and speed the development of PCA applications, the MSI includes a standard method for describing metadata contexts, as well as a standard method for expressing and storing metadata. For example, metadata is stored in XML text files. The PCA metadata system is fully described in the document titled “PCA Metadata System” [13].

Machine Model Metadata Context

A PCA machine model is a generic model used to describe specific PCA platforms. The primary purpose of the machine model is to describe PCA platforms in a common manner to the various tools in the MSI framework. The consumers of this description are high-level compilers, which are responsible for dividing applications into appropriately sized portions for a target platform. The parameters of these models are communicated to the compilers and other tools as metadata in the machine model metadata context, which is formalized in the document titled “PCA Machine Model” [14].

A PCA machine model contains three types of elements: *resources*, *ingredients*, and *morphs*. A resource is a description of a discrete device with well-defined functionality. There are three kinds of resources in a machine model. These are:

- *Processors*: anything that consumes, produces, or moves data. Examples of processors are functional units, DMA engines, and I/O devices.
- *Memories:* Any resource that stores data. Examples are RAM, FIFOs, and cache memory.
- *Network Links:* Any communication pathway between resources. Network links may have an arbitrary number of senders and receivers.

Resources have a variety of parameters to allow detailed descriptions of their capabilities.

An ingredient is a description of an arbitrary, configurable division of underlying hardware. An ingredient can be configured, possibly in concert with other ingredients, into one or more resources. Ingredients provide a mechanism for identifying resources that cannot coexist on the physical platform.

Finally, a morph in the machine model is a description of a state into which a PCA platform may be configured. The metadata describing the morph defines how the machine model ingredients can be configured into resources by the compiler. More than one morph may be active at once, but no two morphs that share an ingredient may be active at once.

CREATING AN APPLICATION

The general flow for creating a PCA application using the Morphware Stable Interface is illustrated in Figure 9. Application programmers using the MSI first partition their application according to the high-level language that will be used to express their algorithms. Parts that fit the streaming paradigm are written in Stream or Brook, while non-streaming parts may be written in C or C++. The high-level language code, augmented with application performance metadata and a machine model metadata description of the target architecture, is compiled by a high-level compiler to SAAL code, which is based on the SVM and UVM APIs, respectively. The HLC may be provided by a PCA architecture vendor or by a third party. Reservoir Labs, Inc.'s R-Stream is an example of a third-party HLC [11]. Application performance metadata must be provided by the application programmer, while the machine model is supplied by the architecture vendor. The application source code may also call functions in libraries that were written by experts using the UVM or TVM-HAL APIs directly.

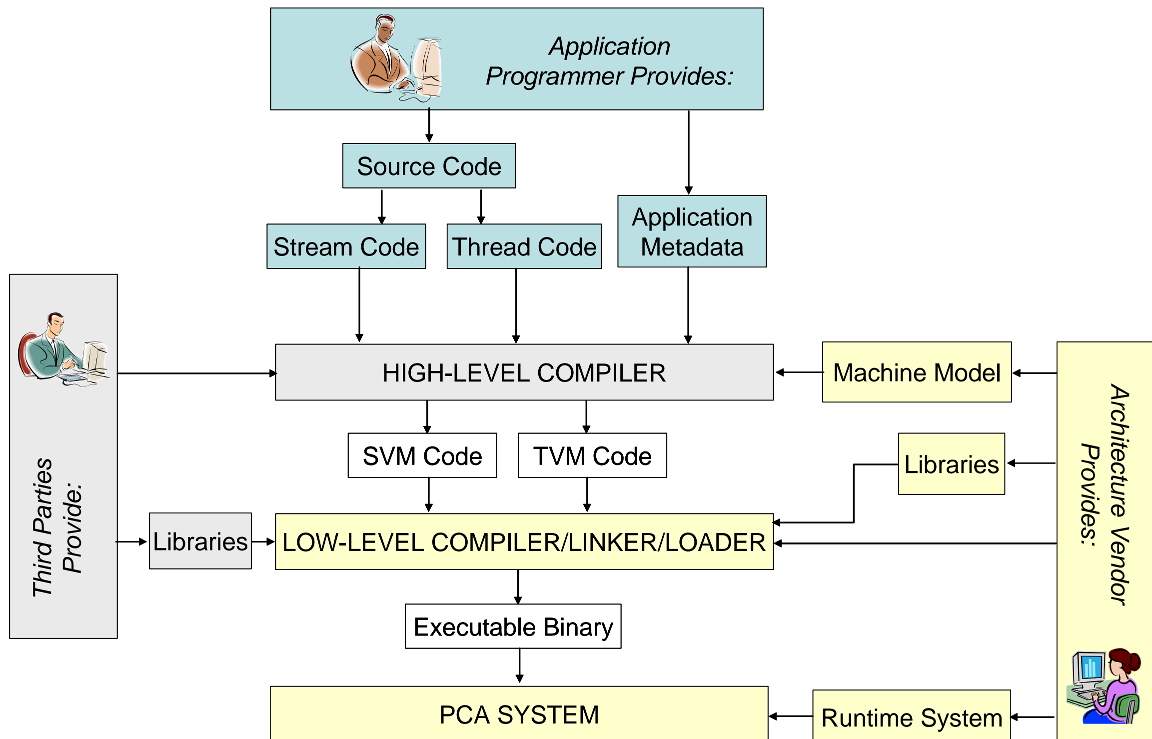

Figure 9. Methodology for creating a PCA application using the MSI.

Each architecture vendor provides one or more low-level compilers specific to their PCA system or device architecture. The various files in the SAAL intermediate formats are compiled for the target architecture, producing architecture-specific binary files. These are linked as appropriate to create one or a set of binaries that are ready to be executed. These binaries are annotated with a set of hardware resource requirements needed for the program to successfully execute. When the program is executed, the PCA run-time system on the target verifies that the required resources are available, allocates the

resources and assigns them to the executing program, and configures the PCA hardware into the initial configuration before starting and running the program. During execution, the hardware may be reconfigured by the application, or by the system responding to changes in the run-time environment.

More detailed and specific information on the process of creating and executing an application using PCA hardware and the MSI is available in the references mentioned elsewhere in this document as well as at the web sites of the various PCA architecture providers.

FUTURE DEVELOPMENTS

To date, the Morphware Forum has focused on development of the two-level compile structure and machine model, thus creating a portable development process for compiling basic program units and supporting libraries. However, these elements of the MSI are not sufficient to meet the full set of goals of the PCA program. The Forum is working actively to identify and prioritize the additional services, APIs, and metadata contexts needed for a fully functional PCA system, and to extend the MSI to include them. Extensions that have been discussed include:

- New extensions to program loaders and linkers to support selection and composition of program components from multiple options;
- New operating system (OS) services and calls to support the unique requirements of application morphing at the various levels described in Figure 2;
- A portable resource management system to manage the real-time selection and control of alternative program instantiations;
- A component-based application software architecture; and
- A composable development system that can provide a stable, portable interface to the application programmer while still being customized to the unique lower-level development environment for each PCA architecture.

Once implemented, these MSI services and architectures will support the advanced capabilities made possible by PCA devices. Most fundamental will be the capability to implement morphing across the full spectrum of circumstances detailed in Figure 2. This in turn is an essential capability to support important end-user system attributes such as validation and verification (V&V) of PCA applications, and real-time fault tolerance in mission-critical applications. For instance, V&V is reported to account for as much as 40 to 70% of system costs [20]. The PCA program is seeking to implement a significant portion of the V&V capability in the application build cycle by extending standard techniques with run-time monitoring and checking. Such an approach would likely be significantly integrated with the run-time resource management portion of the MSI. Similarly, the morphing capability of PCA technology offers significant new flexibility in the system response to run-time faults. Utilizing this capability effectively will take advantage of the component-based approach to PCA software and will again require tight integration with the resource management functions.

REFERENCES

1. The Polymorphous Computing Architectures (PCA) Program, Information Processing Technology Office (IPTO), Defense Advanced Research Projects Agency (DARPA), www.darpa.mil/ipto/Programs/pca/index.htm.

2. The PCA Morphware Forum, www.morphware.org.

3. M. B. Taylor *et al*, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," IEEE *Micro*, March-April 2002. See also cag.lcs.mit.edu/raw/.

4. K. Mai *et al*, "Smart Memories: A Modular Reconfigurable Architecture," *Proceedings* of the 27th Annual International Symposium on Computer Architecture, June 2000. See also www-vlsi.stanford.edu/smart_memories.

5. J. Granacki and M. Vahey, "MONARCH: A Morphable Networked micro-ARCHitecture," presentation to High Performance Embedded Computing Workshop, October 2002. See also www.isi.edu/asd/monarch/.

6. S. W. Keckler *et al*, "A Wire-Delay Scalable Microprocessor Architecture for High Performance Systems," *International Solid-State Circuits Conference (ISSCC)*, pp. 1068-1069, February 2003. See also www.cs.utexas.edu/users/cart/trips/.

7. "Morph Taxonomy," Georgia Institute of Technology and SPAWAR Systems Center San Diego, white paper dated Feb. 3, 2004. Available at www.morphware.org.

8. B. Coate, J. Lebak, J. McMahon, and A. Reuther, "Basic Deliverable of the Integrated Radar-Tracker Application," MIT Lincoln Laboratory, February 21, 2003. This document in turn references more detailed design documents for each major IRT component. Available at www.morphware.org.

9. "StreamIt Language Specification", Version 2.0, MIT Computer Science and Artificial Intelligence Laboratory, October 17, 2003. Available at cag.lcs.mit.edu/streamit/papers/streamit-lang-spec.pdf.

10. P. Mattson, E. Schweitz, M. Engle, V. Litvinov, and K. Mackenzie, "R-Stream 1.0 Brook Clarification", Reservoir Labs, November 5, 2003.

11. R-Stream 1.0 release package, Reservoir, Inc., November 2003. Package may be requested via the Morphware Forum web site, www.morphware.org.

12. Stanford University "Streaming Languages" web page, graphics.stanford.edu/streamlang/.

13. "The PCA Metadata System", v. 0.9, Georgia Institute of Technology and SPAWAR Systems Center San Diego, Jan. 28, 2004. To be available at www.morphware.org.

14. "PCA Machine Model," September 17, 2003. Available at www.morphware.org.

15. "PCA Morphware Virtual Machine Specification," November 4, 2003. Available at www.morphware.org.

16. Lance Hammond, "TVM-HAL Specification", Smart Memories Group, Stanford University, June 2003. Available at www.morphware.org.

17. "User-level TVM (UVM) Interface Specification," September 3, 2003 revision, TRIPS Project, Department of Computer Sciences, The University of Texas at Austin. Available at www.morphware.org.

18. T. Anderson, B. Bershad, E. Lazowska, and H. Levy. "Scheduler activations: Effective kernel support for the user-level management of parallelism", ACM *Transactions on Computer Systems*, vol. 10(1), pp. 53-79, February 1992.

19. S. Amarasinghe and W. Thies, "Stream Languages and Programming Models," presentation at PACT 2003, Sept. 27, 2003. Available at www.cag.lcs.mit.edu/streamit/talks/pact03tutorial/pact03languages.pdf.

20. M. Amduka, D. Krecker, and O. Sokolsky, "Run-Time Environment and Design Application for Polymorphous Technology Verification and Validation (READAPT V&V)", presented at Morphware Forum meeting, December 10, 2003. Available at www.morphware.org.

APPENDIX: STREAM LANGUAGE EXAMPLES

Brook Example

The Reservoir, Inc. R-Stream 1.0 release contains several sample source codes, including a simple Brook program that sums all of the elements of two streams [11]. Figure A-1 is a block diagram of that program; the program code follows.

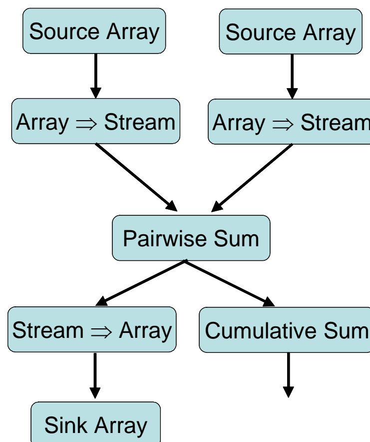

Figure A-1. Block diagram of example Brook program for pairwise sum of two streams.

```
/***---**--*
 *
 *  Copyright (C) 2003 Reservoir Labs. All rights reserved.
 *
 *--**---***/


/*-----------------------------------------------*
 * Stream Sum                                    *
 *                                               *
 *   Get the sum of all the elements in a stream *
 *-----------------------------------------------*/


extern int printf(char [], ...);


/*----------------------------------------------------------------
  Simple kernel which computes the pair-wise sum of two streams
  of elements. For example, the 5th element of the output stream
  c is the sum of the 5th elements of the input streams a and b.
  ----------------------------------------------------------------*/
kernel void pairwiseSum(int a<>, int b<>, out int c<>) {
  c = a + b;
}
```

```
/*----------------------------------------------------------------
  Simple kernel which computes the total of a stream of elements.
  Specifically, the reduction value total is set equal to the sum
  of all elements in the input stream a.
  ----------------------------------------------------------------*/
kernel void computeTotal(int a<>, reduce int total) {
  total += a;
}

/*----------------------------------------------------------------
  Brook code which reads the contents from arrays a_array and
  b_array into streams a and b using the streamReadAll stream
  operator, applies pairwiseSum kernel to those streams, and
  writes the output stream c back to the array c_array using the
  streamWriteAll stream operator.

  Also uses the computeTotal kernel to compute the total of all
  elements in c and prints the result.
  ----------------------------------------------------------------*/
int main(void) {
  int i;
  int total = 0;
  int a_array[100];
  int b_array[100];
  int c_array[100];
  int sum = 0;
  int a<>, b<>, c<>;

  for(i = 0; i < 100; i++) {
    a_array[i] = i;
    b_array[i] = 100 + i;
    sum += i + 100 + i;
  }

  streamReadAll(a, a_array);
  streamReadAll(b, b_array);

  pairwiseSum(a, b, c);

  streamWriteAll(c, c_array);

  computeTotal(c, total);

  printf("\n\nReal sum = %d\n", sum);
  printf("Calculated sum = %d\n", total);

  if(sum == total) {
    printf("PASSED!\n\n");
  } else {
    printf("FAILED!\n\n");
  }

  return (total != sum);
}
```

StreamIt Example

The following example of a frequency band detector, consisting of an A/D converter followed by a bandpass filter and four detectors, is taken from [19]. Figure A-2 shows a block diagram of the system; StreamIt code to implement the system follows.

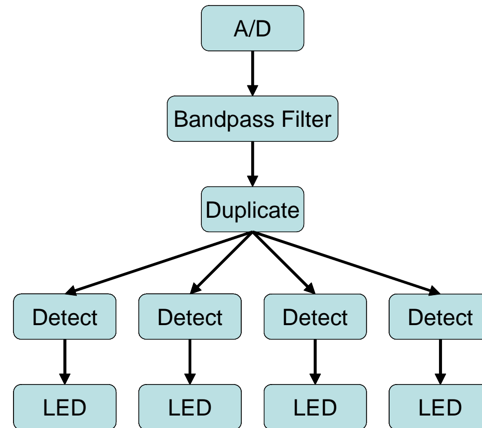

Figure A-2. Block diagram of frequency band detector. (After [19].)

```
void -> void pipeline FrequencyBand {
   float sFreq = 4000;
   float cFreq = 500/(sFreq*2*pi);
   float wFreq = 100/(sFreq*2*pi);

   add BandPassFilter(1, cFreq-wFreq, cFreq+wFreq, 100);

   add splitjoin {

      split duplicate;
      for (int i=0; i<4; i++) {
         add pipeline {
            add Detector(i/4);
            add LEDOutput(i);
         }
      }
      join roundrobin(0);
   }
}
```

As an example of a kernel, the following code implements a low pass filter:

```
float -> float filter LowPassFilter (int N, float freq) {
   float[N] weights;

   init {
      weights = calcWeights(N, freq);
   }

   work push 1 pop 1 peek N {
      float result = 0;
      for (int i=0; i<weights.length; i++) {
         result =+ weights[i]*peek(i);
```

```
            }
            push(result);
            pop();
        }
    }
```